

UNIVERSITY OF ALBERTA

Developing CSMA/CD Protocols with LANSF Observers

by

Marcel Joseph Berard

A Thesis

Submitted to the faculty of Graduate Studies and Research in partial fulfillment of the
requirements for the degree of Master Of Science

Department of Computing Science

Edmonton, Alberta

Spring 1991



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-66702-8

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Marcel Joseph Berard

TITLE OF THESIS: Developing CSMA/CD Protocols with LANSE Observers

DEGREE: Master Of Science

YEAR THIS DEGREE GRANTED: 1991

Permission is hereby granted to the UNIVERSITY OF ALBERTA LIBRARY to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) *Marcel J. Berard*

Permanent Address:

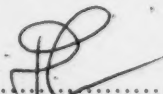
#11, 11108 - 108 Ave.
Edmonton, Alberta
Canada T5H 3Z3

Date: December 18, 1990

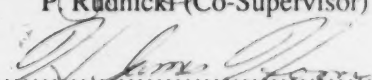
UNIVERSITY OF ALBERTA

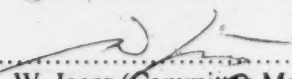
FACULTY OF GRADUATE STUDIES AND RESEARCH

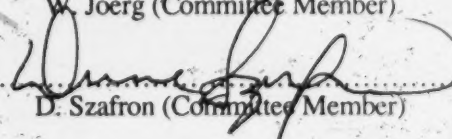
The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Developing CSMA/CD Protocols with LANSF Observers** submitted by **Marcel Joseph Berard** in partial fulfillment of the requirements for the degree of **Master Of Science**.


.....
P. Gburzynski (Co-Supervisor)


.....
P. Rudnicki (Co-Supervisor)


.....
J. Hoover (Committee Member)


.....
W. Joerg (Committee Member)


.....
D. Szafron (Committee Member)

Date: October 12, 1990

Dedication

This thesis is dedicated in memory of a young friend, Robyn Lynn Knorr, who passed away this summer at the tender age of six years after a short bout of encephalitis. She has departed from this earth but has not been forgotten.

Even though I go through the deepest darkness,

I will not be afraid, Lord,

for you are with me.

Psalms 23:4

Abstract

We propose a development methodology for multi-station Medium Access Control (MAC) protocols on broadcast channels. We suggest that standard Open System Interconnection Formal Description Techniques and their supporting tools are poorly suited to specify and validate protocols for this sublayer; the main reasons being the lack of a channel description and the state space size. The correctness of these protocols must therefore be established by a formal, inductive proof over the state space. To simplify this proof, the state space is partitioned into classes. The states in each class are equivalent with respect to a global protocol property. This partitioning is protocol dependent, it does not lend itself to a mechanical derivation. We discuss a software environment to facilitate the process of discovering an appropriate partition that is tested by random state exploration using multiple protocol views. The first view is the implementation, the second is an observer executing in parallel and monitoring the implementation. Our case study develops a Carrier Sense Multiple Access with Collision Detect (CSMA/CD) protocol using synchronized retransmissions.

Acknowledgements

The author has many people to thank for their contributions to this thesis. Two people that come to mind immediately are my co-supervisors Pawel Gburzynski and Piotr Rudnicki. Pawel is the author of the simulation facility used for the thesis research and was always helpful in answering my questions regarding its operation. Piotr suggested the project that is written up in this document and was the willing victim when somebody had to read and criticize previous versions of my manuscript. I thank them both for their contributions. In addition, I would like to commend and thank the other people on my examination committee; Jim Hoover, Dwayne Szafron, Werner Joerg and the chairman Randy Goebel. Their questions showed a genuine interest in the work performed here and suggested some final improvements to this thesis. There are many others who have contributed along the way and rather than trying to name them individually, I'll conclude with a collective thank you to everybody not explicitly mentioned.

Table Of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. Open Systems Interconnection Network Architecture	4
1.2. Formal Description Techniques.....	6
1.3. Protocol Simulation.....	8
1.4. Conformance Testing.....	11
1.5. Thesis Overview.....	13
Chapter 2: CSMA/CD Protocols	15
2.1. History of CSMA/CD Protocols.....	16
2.2. OSI Standards for CSMA/CD Protocols.....	19
2.3. Two Sample CSMA/CD Protocols.....	21
Chapter 3: Local Area Network Simulation Facility	25
3.1. An Overview of LANSF.....	26
3.2. Communicating Finite State Machines.....	28
3.3. The LANSF Specification Language.....	29
3.4. An Ethernet Specification.....	30
Chapter 4: Tree Collision Resolution	33
4.1. Slotted Semi-Controlled Protocols.....	35
4.2. Slotted Tree Collision Resolution.....	36
4.3. LANSF Specification of CSMA/CD-TCR.....	40
Chapter 5: Implementation Observers	44
5.1. Implementation Conformance Testing.....	45
5.2. CSMA/CD Observers.....	50
5.3. LANSF Observers - Model and Specification.....	52
Chapter 6: Verifying CSMA/CD-TCR	56
6.1. CSMA/CD Observer States/Transitions.....	57
6.2. A Sample Observer.....	60
6.3. Results of Observer Conformance Testing.....	64
Chapter 7: Modelling Semi-Controlled CSMA/CD Protocols	68
7.1. The Uncontrolled Mode of CSMA/CD Protocols.....	69
7.2. Slot Synchronization for Semi-Controlled Protocols.....	71
7.3. The Controlled Mode of CSMA/CD Protocols.....	74
7.4. Timing Correctness of Controlled Mode Slots.....	76

Chapter 8: LANSF CSMA/CD Implementations	79
8.1. LANSF Normal Mode Implementation.....	81
8.2. LANSF Controlled Mode Implementation.....	83
8.3. The DP Protocol Implementation.....	85
8.4. The TCR Protocol Implementation.....	86
8.5. Simulation Results with the TCR Implementation.....	88
Chapter 9: Conclusions	91
9.1. Random State Exploration.....	93
9.2. Future Directions.....	94
Bibliography	97
Appendix A: Services used by station processes in CSMA/CD protocols	100
Appendix B: LANSF Ethernet transmitter	102
Appendix C: LANSF CSMA/CD-TCR transmitter (Initial implementation)	103
Appendix D: LANSF CSMA/CD-TCR observers	105
Appendix E: LANSF CSMA/CD-TCR transmitter (Final implementation)	107

List Of Figures

Figure		Page
1.1	OSI network layers.....	5
1.2	Observer configurations.....	12
2.1	A CSMA/CD network.....	15
2.2	Slotting time in a network.....	16
2.3	CSMA/CD sublayers in the OSI model.....	19
2.4	Ethernet retransmission mode.....	21
2.5	Dynamic priority retransmission mode.....	22
3.1	A Communicating Finite State Machine.....	28
3.2	CSMA/CD receiver states/transitions.....	31
3.3	Ethernet transmitter states/transitions.....	31
4.1	A slotted DP time line.....	35
4.2	A sample TCR virtual binary tree.....	36
4.3	A slotted TCR tournament.....	38
4.4	A tournament tree for slotted TCR.....	39
4.5	CSMA/CD-TCR normal mode.....	41
4.6	CSMA/CD-TCR retransmission mode.....	42
5.1	Local conformance test system architecture.....	46
5.2	Distributed conformance test system architecture.....	47
5.3	States/transitions of a LANSF observer.....	53
6.1	Two views of CSMA/CD uncontrolled mode.....	57
6.2	CSMA/CD observer slot states.....	58
6.3	CSMA/CD observer collision states.....	59
6.4	Observer vs. IUT view of slots.....	66
7.1	CSMA/CD uncontrolled mode.....	69
7.2	Different station views of slots.....	71
7.3	Slot with guard intervals.....	73
7.4	CSMA/CD controlled mode.....	75

Chapter 1

Introduction

With the number of computers increasing rapidly, communication networks are becoming very common. Communication protocols for these networks are based on distributed algorithms. The non-deterministic, parallel execution of protocols magnifies the problem of ensuring software correctness relative to traditional sequential programs. Existing research indicates various techniques to specify these protocols and ensure their correctness. Before reviewing existing techniques, it is useful to define relevant terminology. The literature has variations, most of what follows conforms to Sunshine [SUNS79]. A *protocol entity* is a party communicating according to protocol rules, it is equivalent here to a network station. The *service specification* abstractly and informally defines services provided by protocol entities; this should match services desired by potential users. An *interface specification* dictates how users request services at the *service access points* (SAPs) of protocol entities. The *protocol specification* abstractly describes protocol entities, that is, the interactions between them to provide requested services. Protocol specifications only provide essential information (ie. messages exchanged by entities) while leaving out implementation dependent details. There are various formal methods to specify protocols which include finite state machines (FSMs), Petri nets, algebraic calculi, high-level programming languages, abstract data types and temporal logic, see [BOCH90] for a recent introduction to this topic. The complete internal description of protocol entities is an *implementation* of the protocol. An implementation can be alternately viewed as an *executable specification*. This thesis considers communicating extended finite state machines for the implementation of protocol entities.

The process of ensuring protocol software correctness is called *protocol engineering* [RUDI88]. Protocol correctness can be divided into two aspects [SUNS79]. To start, (non-terminating) communication protocols must meet general properties such as:

- 1) Freedom from deadlock
- 2) Completeness
- 3) Stability
- 4) Progress

This aspect can be viewed as the syntactic component of protocol correctness [SUN79]; ensuring its correctness is called *protocol validation*. General properties can be validated by protocol state exploration. Freedom from deadlock means the protocol cannot enter a state without an exit, every state satisfies the predicate $P = \text{"There exists an exit from this state"}$. Completeness infers that states anticipate all possible incoming messages (ie. no unspecified receptions); a specification without completeness is *underspecified*. The opposite error is *overspecification* in which a state anticipates a message that cannot arrive. This may seem irrelevant since it cannot create a run time error. However, such errors indicate a protocol design error. The worst overspecification case is when all anticipated messages cannot arrive, this defines a deadlock state. Specification stability is the robustness to resume some form of normal behavior following abnormal events such as hardware failure. Progress (ie. no livelocks) means that useful work is performed by the network. It appears that general properties are dynamic since they involve state analysis. However, they can be validated from a textual specification, that is a state model can be built using a generic tool. The key point is that these properties can be validated without an implementation.

The second aspect of ensuring protocol correctness relates to semantic conformance and is called *verification*. This conformance means the specification or implementation provides functions specific to that protocol. In analogy with sequential programs, protocol semantic correctness is more difficult to prove than syntactic correctness. Semantic correctness involves not only a static analysis of states but their dynamic behavior. There are two basic verification techniques, formal (algebraic) proofs or exhaustive state space exploration. The algebras of most specification techniques are not sufficiently understood and/or powerful enough to provide formal correctness proofs. Formal proofs are further complicated by protocol event non-determinism which precludes complete knowledge of successor states.

The existing literature on semantic correctness proofs leans heavily towards exhaustive (or partial) state space exploration. The concept is simple, if each reachable state of a protocol satisfies a predicate P ,

then P is a protocol property. This technique is also called *reachability analysis*, which means that states reachable from the initial state are analyzed. In practice, exhaustive explorations often require excessive computer resources and are not performed. The alternative is partial searches in which protocol correctness probability is maximized for a given amount of resources. Some methods used for this "maximization" are considered later in the chapter. From this discussion, verification state searches appear identical to those of validation. They are conceptually similar, however verification is not well suited to generic tools and typically requires an implementation. To illustrate, a protocol may guarantee packet delivery within a period τ (a service specification). Protocol conformance or non-conformance to such a property is difficult to prove with generic tools since they may not reproduce implementation timing properties. In effect, predicates P for semantic properties are more complex than those of syntactic properties. For a recent article on tools used to validate and verify properties of specifications, see [PEHR90].

The testing of implementations by state exploration is called *implementation verification* or *conformance testing*. Conformance testing can verify conformance to the protocol specification which confirms the translation process from specification to implementation. Conformance testing may also verify specification correctness. In this situation a correct translation is assumed; the implementation is simply another state exploration tool to test syntactic or semantic properties of the specification. Explorations usually involve state sequences that test protocol specifications under stressful conditions. Some languages have been designed to express abstract test sequences. When there is no protocol specification, conformance testing directly verifies implementation conformance to service properties. This testing situation is less rigorous as service specifications are usually informal.

The balance of this chapter expands on existing methods of protocol engineering. Network protocols are usually specified and verified on a layered basis; the first section of this chapter deals with protocol layering in the Open Systems Interconnection (OSI) model. Section 2 examines the use of Formal Description Techniques (FDTs) to specify communication protocols. The third section examines how simulation results can be used as a protocol testing tool. After this follows a general overview of conformance testing in simulated and real environments. The last section considers how this thesis relates to existing protocol engineering techniques and its new contribution to this area.

1.1. Open Systems Interconnection Network Architecture

The most common network architecture is the Open Systems Interconnection (OSI) reference model based on an International Standards Organization (ISO) proposal [ISO82] to facilitate connections of open systems. A review of this proposal can be found in [DAY83]. OSI has been well accepted as a standard in some areas but not in others, a summary of its acceptance is presented in [CARP89]. In OSI, the key concept is protocol layering which is implemented at every protocol entity. The underlying technique is divide and conquer as used in traditional programming. That is, to simplify a complex protocol break it into layers with clean interfaces. For protocols, each successive layer abstracts higher level concepts (services) by building on services of the previous layer. With appropriate layering, complex services are provided through multiple but relatively simple primitives. An additional OSI benefit of layering is that it facilitates the interfacing of independent implementations at adjacent layers. The OSI model specifies seven protocol layers, these are:

- 1) Physical layer
- 2) Data link control layer
- 3) Network layer
- 4) Transport layer
- 5) Session layer
- 6) Presentation layer
- 7) Application layer

This layering is shown in figure 1.1, the communication medium is controlled by the physical layer. Users (virtual layer 8) request services from the application layer. Bold vertical lines symbolize interfaces between adjacent layers at a station. Lines joining layers n and $n+1$ correspond to service access points of layer n . These interfaces specify services provided by layer n and used by layer $n+1$.

Figure 1.1 also shows horizontal interfaces between peer entity processes. Communications between layer n peers are defined by layer n protocols. These protocols ($n > 1$) are viewed as direct message exchanges but are implemented as service requests to layer $n-1$. Only the physical layer has direct transfers, its horizontal interfaces are physical interconnections. Horizontal interfaces at higher layers are virtual

connections as indicated by dashed lines. The number of peers at a layer is an important factor. The Transport to Application layers generally have two peers per virtual channel; a separate channel exists for each communicating pair. *Protocols for these layers specify the behavior of two communicating entities.* Lower layers can have an arbitrary number of peers depending on channel hardware and network topology. *Some protocols for layers 2 and 3 must account for a greater and variable number of communicating entities.* This thesis examines data link layer protocols that are burdened with this difficulty.

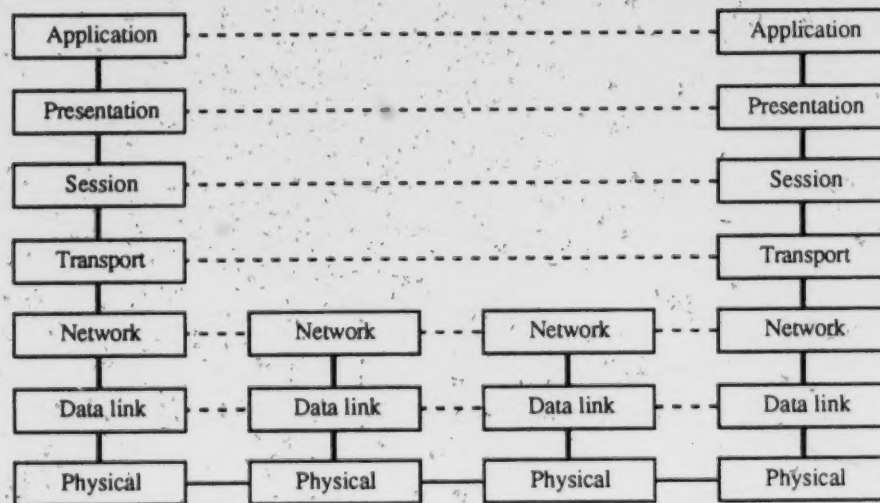


Figure 1.1 - OSI network layers

Recall that verification techniques explore protocol states to determine correctness. For layers 4 to 7, there exist autonomous channels for each pair of communicating entities. Thus, the global state Φ is a product of independent (orthogonal) channel states $\Phi = C_1 \dots C_K$ where K is the number of channels and C_i the state of channel i . Each channel state depends on the local states of two entities, $C_i = \phi_0 \phi_1$ where ϕ_j is a local state. If each entity has s internal states, a channel has at most s^2 states. In this case, it may be possible to formally verify a protocol specification by exhaustive channel state space analysis. If this is not possible, key portions of the state space can be explored to demonstrate partial correctness. A proof that two peers (partially) operate correctly in a channel proves (partial) protocol correctness for that layer.

This thesis examines Medium Access Control (MAC) protocols in the data link layer. The OSI refinement for this protocol class is examined in the next chapter. The studied subclass is Carrier Sense, Multiple Access with Collision-Detect (CSMA/CD). CSMA/CD protocols control access to a shared medium (physical channel) on a demand basis. The number of stations sharing the channel is not defined a priori, such protocols are required to cope with this feature. With CSMA/CD protocols, the global state cannot be factored into orthogonal two-entity channel substates. Every communication is broadcast; it is heard by and affects all stations. The global state is a product of all local states; $\Phi = \phi_0 \dots \phi_{x-1}$ where x is the number of stations. The maximum number of global states now increases to s^x . An increasing number of stations produces an intractable state explosion which precludes exhaustive state space analysis. To complicate matters further, a proof of protocol correctness for a specific value of x does not demonstrate correctness for a general number of stations. This protocol class requires new techniques for global state description to permit verification.

1.2. Formal Description Techniques

To facilitate the *Open* in OSI, protocol specifications should permit a consistent interpretation by different parties wishing to implement the protocol. This requires a precise and well known specification language. In the early days of communication protocols (1970s), specifications were informal (natural language) and tended to produce interpretation ambiguities. Implementation languages were standard procedural languages which permitted further ambiguities and/or errors in the manual translation from specification to implementation. Any analysis of these specifications was also performed manually. Correctness proofs were related to formal interpretations placed on informal specifications. During this era there were few attempts to connect different implementations of the same protocol in a network. This process which is called *interworking* implementations is an obvious goal of OSI.

The 1980s saw a rapid proliferation of computer networks and dictated a need for interworking implementations. This prompted the emergence of Formal Description Techniques (FDTs) to specify protocols. FDTs include the well known OSI languages LOTOS [LOT089] and Estelle [ESTE89] as well

as the CCITT language SDL [SDL88] which have been proposed as standards. Introductions to these languages can be found in [BOLO87], [BUDK87] and [BELI89] respectively. These languages have formal constructs to specify communication protocols and their precise semantics permit rigorous analysis. These languages have reduced but not eliminated specification ambiguities because of incompleteness in their definitions. There is also the problem of differences between specification and implementation languages; nuances can be lost in a translation.

With FDTs formal algebraic verifications of complex protocols have not been performed; the typical analysis consists of mechanical state searches. When exhaustive searches are infeasible, selective searches are performed. These searches explore states reached from the initial state by a collection of "difficult" event sequences called a *test suite*. There exist abstract languages such as *Tree Table Combined Notation* (TTCN) to express test suites. A test suite determines the truth or falsity of predicates P_i = "This specification is correct for event sequence i ." However, this does not prove P = "This specification is correct." The correctness problem then returns to human experience, in knowing where to look for specification errors. A test suite forms a collection P_i that "almost" proves P , in the sense that networks should operate for long periods before producing conditions that did not occur in some of the P_i .

Despite some limitations FDTs have demonstrated their usefulness with OSI protocol specifications, particularly for layers 4 to 7 and to a lesser extent for layers 2 and 3. FDTs are well-suited to two party protocols with temporal relationships such as "event A precedes event B". The first table in [BOCH90] provides a list of FDT specifications for OSI communication protocols and services. There is considerable research in validation and verification techniques for these specifications. Further, to reduce ambiguities in producing implementations, this task has been partially mechanized. Sidhu and Blumer [SIDH90] provide an introduction to the current state of semi-automatic OSI protocol implementation. The field of conformance testing for FDT implementations also has mechanical support. Some authors have worked on compiling abstract test sequences to FDTs or other executable languages, see [LINN90] for a survey.

In comparison, there is a dearth of formal specifications for the Medium Access Control sublayer. The problem with FDTs at this level is their weak support for modelling signal propagation in communication channels; a critical component of MAC protocols. There has been some effort to specify

these protocols with formal languages. Parrow [PARR88] attempted to specify asynchronous CSMA/CD (eg. Ethernet) with CCS (Calculus of Communicating Systems) and concluded: "*Within CCS, it is impossible to describe properties related to efficiency, throughput and other aspects related to time. Also, it is impossible to describe fairness properties and to prove absence of livelocks.*" In describing the state space, he limited his analysis to two parties for simplicity. An attempt was also made by P. Rudnicki at the University of Alberta to specify CSMA/CD with CSP (Communicating Sequential Processes); it lead to similar conclusions. The problem encountered by both authors relates to describing signal propagation using the FSM notation, this involves precise delays in transporting messages, collisions, etc. Even if CSMA/CD protocols could be specified with FDTs, there would still be verification problems because of the number of protocol parties. As stated earlier, most FDT verification tools use state space exploration. With 2 party protocols this is plausible but for a large and variable number of parties these tools are not effective. The state space of CSMA/CD is essentially infinite; its verification requires an inductive technique, not the brute force of exhaustive state exploration. The conclusion is that FDTs are useful for most protocol specifications but MAC protocols are a noticeable exception.

1.3. Protocol Simulation

The previous sections described techniques to facilitate correct protocol design. There is yet another tool available before a hardware implementation is created. It is well known that computer simulations are often more cost effective than hardware prototypes. Local Area Network Simulation Facility (LANSF) [GBUR89a, GBUR89b] applies this principle to communication protocols. Simulation is conformance testing that uses a virtual implementation to perform state exploration. It rates a separate section here because of its importance within this thesis. With LANSF simulations, it is possible to investigate not only protocol correctness but also performance. Initially, MAC protocols were simulated with LANSF to study their performance. The success in reaching this goal can be seen from the literature, some examples include [DOBO88] and [DOBO89]. The advantage of this approach compared to analytic modelling can be condensed from the LANSF documentation [GBUR89b]:

A mathematical model of the network is built and a formal analysis of that model is carried out. Unfortunately, exact and tractable mathematical models exist for very few networks and protocols - in most cases, one has to put up with some simplifications. These simplifications make the model tractable, but at the same time, they affect its accuracy: the model does not reflect the reality exactly and some details are lost.

Cases when the impact of the simplifying assumptions on the accuracy of the model is investigated or at least discussed are rather scarce. Every simplification introduces an essentially immeasurable amount of uncertainty into the research.

For the most part this criticism also applies to FDT specifications; they provide no indication of protocol performance. However, performance analysis was not an initial OSI goal. This issue has received some attention recently, for example [BOCH88]. Rudin [RUDI88] summarizes this growing concern:

At the same time, interest in the performance of protocols, particularly the OSI family of protocols, is rising sharply. There is a growing concern that straightforward implementation of the OSI protocols will restrict throughput to a small fraction of what the underlying transmission technology would allow.

In contrast, the problem of implementation correctness for LANSF MAC protocol research has received little attention. One weakness is the lack of a formal specification language. Informal MAC protocol specification precludes formal verification prior to implementation. There can also be ambiguities when translating from an informal specification. A typical LANSF methodology for ensuring correctness has been the following: Program the protocol (implementation) until simulation throughput results seem reasonable, then assume the implementation is correct. In the author's opinion, this method although not rigorous, is not without merit. For MAC protocols, it indicates some errors serious enough to produce a lock (deadlock or livelock) with long but feasible random state explorations. Stated another way it validates some general protocol properties.

The author has observed that protocol locks usually manifest themselves in the performance results of two different simulations. To illustrate, the average throughput for simulations of 100 and 100,000 packets

should statistically differ by something on the order of $\sqrt{100}^{-1} - \sqrt{100000}^{-1}$ or about 10%. Differences beyond this limit suggest a lock after packet 100 but before packet 100,000. For example, a total network lock appearing after 100 but before 50,000 packets should reduce throughput to less than half its expected value for the longer simulation. A lock produced within 100 packets on a long run produces a throughput so close to 0 that the implementation is clearly wrong. Thus, a lock appearing in the first 50,000 packets should be discernable from the throughput of 100 and 100,000 simulated packets. The reason for believing that potential locks occur within 50,000 packets is discussed later. To summarize, the author believes that feasible simulations generally detect fatal MAC protocol errors. However, this optimism should be tempered by a real time calculation for the transmission of 50,000 packets. Assuming an average packet length of 1000 bits on a 10 Mbit/sec network, the simulation covers about 5 sec of real time.

Another potential problem is a partial lock involving a few stations. For example, in a 50 station network, two locked stations decrease throughput by 4% which is within the stated margin. The author's experience with MAC protocols suggests that this situation is unlikely if stations attempt to communicate with all other network stations. There are not separate channels for each communicating entity pair in this sublayer, rather each station has one inward path (input buffer) and one outward path (output buffer). Given the restriction of total interstation network traffic, a lock in any station subset quickly propagates through the entire network as other stations attempt to communicate with a member of the locked subset.

The area of semantic correctness for LANSF protocol implementations has not been well addressed by simulation. It is easy to have minor implementation errors that do not seriously affect throughput in a network of homogeneous implementations (which usually includes simulated protocols). These minor errors can produce major problems in real networks related to interworking different implementations. This can be the legitimate result of arbitrary implementation choices permitted by specifications or possibly logic and coding errors. In fairness, it is not the case that simulation cannot detect interworking errors. The point is that simulations generally use the same protocol implementation at all stations. In simulated environments, interworking gray areas could be located and addressed by insisting on multiple protocol implementations. As will be shown, a different type of multiple specification is exploited by LANSF *observers* described in this thesis.

Protocol simulation is not restricted to MAC protocols with LANSF. There exist simulators for FDT specifications, one example called Véda is described in [JARD88]. This simulation environment executes Estelle specifications. It is of interest here for two reasons; the LANSF specification language is syntactically similar to Estelle and there is research concerning observers in Véda. Simulation in general and with LANSF in particular has shown itself to be a powerful tool. It provides a quick prototype and permits experiments that are difficult to reproduce with physical networks. For example, hardware errors can be simulated to test implementation robustness. The ease of observation permitted in simulated environments also exceeds that of physical networks. Finally, a protocol specified in LANSF can be tested in a friendly environment and then ported to a physical network with very few modifications. This is due to the LANSF specification language (very similar to C) and the fact that it closely reproduces physical environments. The main weakness of simulation is that hardware errors cannot always be reproduced; only real environments show the robustness of protocols.

1.4. Conformance Testing

Conformance testing compares the behavior of an implementation under test (IUT) to some standard. Even for implementations produced from FDT specifications, this is considered an essential step of ensuring correctness. Rayner [RAYN87] states "It is now widely accepted that OSI conformance testing is crucial to the achievement of the objectives of OSI." An introduction to OSI conformance testing can be found in [LINN90]. The standard in conformance testing is typically the protocol specification (if one exists), the service specification or an independent implementation. Testing can be performed in real or simulated environments, the latter provides greater flexibility. The simplest conformance test for a simulated IUT is to manually compare its response to a few events with those expected from the service specification. This technique is similar to print statements used in traditional programming. Extensive conformance testing requires a mechanical method (checking program) that encodes expected behavior. The idea applied here is software redundancy which is multiple independent programs performing identical or related tasks. Molva [MOLV85] describes the role of software redundancy as follows:

If a system is realized with redundant copies, faults occurring in one copy during execution will provoke discrepancy between the behavior of this copy and the behavior of the non-faulty copies. Then to get a correct diagnosis one has to discriminate the faulty copy.

An example of multiple programs performing identical tasks is given by Murphy et al. [MURP89]. The authors describe an experiment in which six implementations were produced with 3 different languages for each of two protocols. These implementations were executed with test data and their output logged. The log files were compared with a harness program to check for differences. In this example, the observable output should be identical. This experiment illustrates the principle of off-line conformance testing.



Figure 1.2 - Observer configurations

Conformance testing can also be performed on-line in real or simulated environments. A process that performs real (simulated) on-line conformance testing is called a *hardware (software) observer*. Both observer types can be either passive or active. Active observers interact with the IUT by sending test sequences and monitoring subsequent output. The corresponding information flow is shown in figure 1.2a. Passive observers monitor but do not affect implementation operation, a possible exception is when they detect an error. This is shown in figure 1.2b, information flows only from IUT to observer. The implementation receives events from another source (incoming line on the left). Observers may or may not independently know of these events as indicated by its incoming dashed line.

The search space examined by the two observer types is quite different. Active observers produce a directed search whereas a passive observer monitors a random search. The test suites described earlier are an example of a directed search. Active observers generally remember explored states by constructing a tree

whose root is the initial state. In this manner, each state is explored at most one time. These observers examine as many states as possible with their given resources. Using test suites the states examined first are those reached by "difficult" event sequences. An alternative proposed by Maxemchuk et al. [MAXE87] first explores the most probable states. In contrast, passive observers generally have no memory. They cannot control the explored states, so there is no point in remembering them to avoid redundant explorations. Random state exploration has the advantages that it explores states rapidly and may discover unexpected errors missed by a directed search. West conducted an experiment with random state searches which is described in [WEST86].

The last observer parameter is the network environment. Hardware observers have the usual advantage of testing implementations in their true environment. An experiment by Molva et al. [MOLV85] used passive hardware observers to test a fault tolerant MAC protocol. These observers successfully detected hardware failures; an impossible task for software observers. Software observers however have many features not available in hardware systems. These include no filtering through lower protocol layers, complete access to IUT variables, total control over simulation parameters and instantaneous knowledge of events. A useful design methodology is to first implement and test a protocol in the friendly software environment and then refine it in the hardware environment.

1.5. Thesis Overview

This thesis describes the use of passive software observers to develop CSMA/CD protocols in the LANSF environment. This class is not easily specified with OSI-FDTs so an alternate protocol design technique is required. The existing LANSF methodology directly implements CSMA/CD protocols from informal service specifications. In the case study of this thesis, three informal service specifications are formalized with observers and used as conformance testing standards. Conformance testing experiments locate two implementation errors relative to these standards. The knowledge of these errors and the observer model are then used to re-implement the protocol. The first goal reached by this thesis is an improved LANSF implementation for the semi-controlled subclass of CSMA/CD protocols that includes the case study.

A second result of this thesis is a simplified formal model for global states and timing properties of semi-controlled CSMA/CD. The observers required a specification that was orthogonal to the implementation. In section 2, it was argued that the local state space of CSMA/CD protocols was too large to verify directly. It is also too complex to reason about protocol properties using this view. Observer global states Φ cannot correspond to the complete global state of network stations ($\Phi_0 \dots \Phi_{x-1}$); a new outlook is required to describe states. The observer model created here partitions the state space into a fixed number of subsets (observer states Φ_i) independent of the number of stations. This is a simplified model in which protocol timing properties can be proven based on the Φ_i . It also provides a framework in which to describe the operation of protocol entities. An example is demonstrated in chapter 7 for timing constraints of network slots within protocol entities.

The thesis outline is as follows: The next three chapters review relevant existing work. Chapter 2 describes general features of CSMA/CD protocols and provides two examples, one with asynchronous and the other with synchronous retransmissions. In the following chapter, the LANSF simulation environment is presented; this includes a sample CSMA/CD protocol implementation. Chapter 4 introduces the case study protocol for this thesis and its initial implementation prior to observer conformance testing. Chapters 5 and 6 describe the observation of the initial LANSF implementation. This starts with a review and general comments about observation as well as the definition of LANSF observers in chapter 5. Chapter 6 presents the observer model and specification used for the case study as well as the experimental results. The next two chapters examine how the protocol can be re-implemented on the basis of knowledge gained from observers. Chapter 7 provides the formal abstract model (specification) and chapter 8 the implementation. The last chapter lists the conclusions of this work.

Chapter 2

CSMA/CD Protocols

This chapter reviews Carrier Sense, Multiple Access with Collision Detect (CSMA/CD) protocols. A well known example of this protocol class is Ethernet[†]. CSMA/CD protocols are used for local area networks with a multiaccess channel. Stations compete for channel access on a demand basis which leads to collisions and packet retransmissions. CSMA/CD protocols attempt to simultaneously satisfy two goals: utilize the channel efficiently and provide equitable service to stations. Conceptually, CSMA/CD stations each have two connections (ports) to the channel. One port is used for input (listening) and the other for output (transmitting). In practice, both logical ports are implemented with a single physical port. A sample network with 4 stations labelled 0...3 is shown in figure 2.1. This figure depicts two packets in the channel, one leaving station 0 and the other traveling from station 2. These packets will collide and require retransmission.

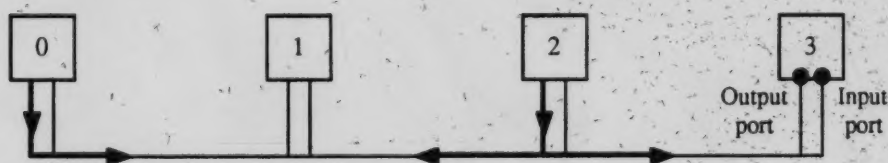


Figure 2.1 - A CSMA/CD network

The first section of this chapter gives a history of CSMA/CD protocols. Section 2 describes how the OSI model is refined for this protocol class. It also examines timing constraints inherent with CSMA/CD. In the last section two sample CSMA/CD protocols are discussed. The first (Ethernet) uses unsynchronized packet retransmissions, the second (Dynamic Priority) uses synchronized retransmissions.

[†] Ethernet is a trademark of the Xerox Corporation.

2.1. History of CSMA/CD Protocols

CSMA/CD starts in 1970 with the ALOHA network at the University of Hawaii [ABRA85]. This network had a central station and multiple peripheral stations. It used a multiaccess (MA) protocol with a radio wave carrier as the common channel. Peripheral stations transmitted as soon as they received a packet, a process called *statistical multiplexing*. This random transmission produced collisions based on statistical factors. Peripheral stations did not immediately detect collisions; the protocol specified confirmation packets from the central station. The lack of a confirmation within a specified period was interpreted as a collision. The ALOHA retransmission algorithm was simple as expected from the first protocol using statistical multiplexing. Damaged packets were retransmitted after a random delay determined locally. The optimum retransmission delay is pulled in opposite directions by two factors. It should be short to minimize packet delay but long to reduce the collision probability between retransmitted packets. One method of approximating the optimum mean delay is considered shortly with the Slotted ALOHA protocol.

At low traffic densities a multiple access channel is usually free and ALOHA provides near optimal performance. There is no wait time and stations use the entire channel bandwidth. Predecessors of ALOHA such as Time Division Multiplexing (TDM) and Frequency Division Multiplexing (FDM) did not have both features. With TDM, stations wait for their turn even if all higher priority stations are idle. In FDM, stations use only a fraction of the total bandwidth. For both TDM and FDM, idle stations restrict the performance of busy stations. The ALOHA weaknesses relative to TDM or FDM are additional overhead and the fact that collisions occur. Collisions increase mean packet delay and reduce maximum throughput. This problem is worse at medium and high traffic densities. One apparent solution mixes the statistical multiplexing of ALOHA at low traffic densities and TDM at high densities. The CSMA/CD-DP protocol discussed later in this chapter presents one version of this mixture.

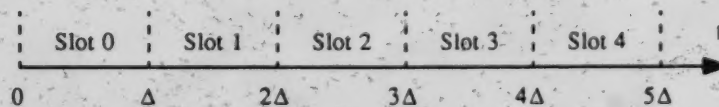


Figure 2.2 - Slotting time in a network

An early modification was Slotted ALOHA [ROBE72]. This protocol divides channel access into slots of length Δ as shown in figure 2.2. In slotted networks transmissions only start on slot boundaries, an idea taken from TDM. Slotted ALOHA transported statistical multiplexing to slotted networks. For an N station TDM network, stations wait an average of $\frac{N}{2}$ slots before transmitting. With Slotted ALOHA new packets are sent during the next slot. If the packet does not collide its delay is on the order of one slot. Slotted ALOHA also has the weakness that collisions occur but it reduces collision frequency relative to ALOHA. To illustrate assume ALOHA packets are constrained to a fixed length Δ . Further, let the interstation distance be negligible. An ALOHA packet started at T remains in the channel until $T + \Delta$ and collides with other packets started between $T - \Delta$ and $T + \Delta$. For a successful transmission there must be no additional packets started during an interval 2Δ . With Slotted ALOHA, a packet arriving in the interval $S_j = [j\Delta, (j+1)\Delta)$ is transmitted during slot $j+1$ and only collides with other packets arriving during S_j . Slotted ALOHA reduces the collision interval to Δ , or half the interval of ALOHA. The consequence is fewer collisions and better performance. At low traffic densities the performance of the two protocols is similar. At medium and high densities the difference is noticeable; Slotted ALOHA has shorter mean packet delays and a higher maximum throughput. ALOHA has a maximum throughput of $\frac{1}{2e}$ or about 18.4%. The corresponding value for Slotted ALOHA is $\frac{1}{e}$ (packets/slot) or 36.8% [TANE88].

Slotted ALOHA provides an insight into the optimum mean delay of random retransmissions. After a collision, sending stations do not know how many packets were involved; a first guess is that two packets collided. Consider an algorithm where two stations randomly retransmit packets in one of the next n slots. The collision probability during retransmission is $\frac{1}{n}$ and the success probability $\frac{n-1}{n}$. During the next n slots, two retransmitted packets have an average throughput of $\frac{1}{n} * 2 * \frac{n-1}{n}$ or $\frac{2(n-1)}{n^2}$ packets/slot. A choice of $n = 2$ maximizes this throughput at 0.5 packets/slot. If a collision involves more than two packets, the optimal value of n is higher. Tanenbaum [TANE88] shows that the maximum throughput of Slotted ALOHA (Poisson arrivals, infinite station population) occurs for an offered load (new and retransmitted packets) of 1 packet/slot. This suggests that n colliding packets should be randomly placed in n slots to maximize their average throughput (ignoring new packets). This result is not exact because it deals with a

finite number of stations, however it illustrates that more slots are needed to efficiently resolve larger collisions. Of course, stations don't know the number of colliding packets but can increase their estimate after each successive collision. One such algorithm is Binary Exponential Backoff (BEB). With BEB, n is set to 2 after the first collision of a packet and doubled for successive collisions. A version of BEB for unslotted networks is presented later in the chapter where Ethernet is discussed.

Stations in ALOHA and Slotted ALOHA networks could not sense the channel status. An improvement is possible in unslotted networks when stations sense the channel (Carrier Sense or CS). In these networks, stations can refrain from using a busy channel, that is they *defer* to existing transmissions. A further improvement is possible if collisions are detected immediately (Collision Detection or CD). In this case, transmissions are aborted and the channel silenced as soon as possible. Carrier Sense and Collision Detection are not useful in slotted networks where transmissions are synchronous and slots completed regardless of their status. The unslotted protocol class with Carrier Sense and Collision Detection is CSMA/CD. A full description is found in the IEEE 802.3 standard [802.3] with similar standards for Token Bus (802.4) and Token Ring (802.5) local area networks. The general service specifications of CSMA/CD protocols are as follows. When a client packet arrives the channel is sensed and if it is busy, stations defer until it becomes idle. If the channel is idle or when it becomes idle an inter-packet gap is enforced. This gap permits the physical layer to separate successive packets. When sending stations hear a collision, they abort transfers immediately. Stations then emit jamming signals to enforce collisions after which they wait until it is time to retransmit. Jamming signals ensure that collisions propagate to and are heard by all network stations. The variations in CSMA/CD protocols relate to retransmitting packets. There are subclasses in which stations retransmit based on an individual asynchronous (uncontrolled) or a collective synchronous (controlled) algorithm. A sample protocol from each subclass is considered in the last section of this chapter.

2.2. OSI Standards for CSMA/CD Protocols

This section describes OSI standardization of CSMA/CD protocols. For this section the precise terms *frame* and *packet* differentiate protocol data units in the data link and network layers respectively. A complete description is found in standard [802.3] which refines the OSI model to isolate features of CSMA/CD protocols. The two lowest OSI layers are split into sublayers as shown in figure 2.3.

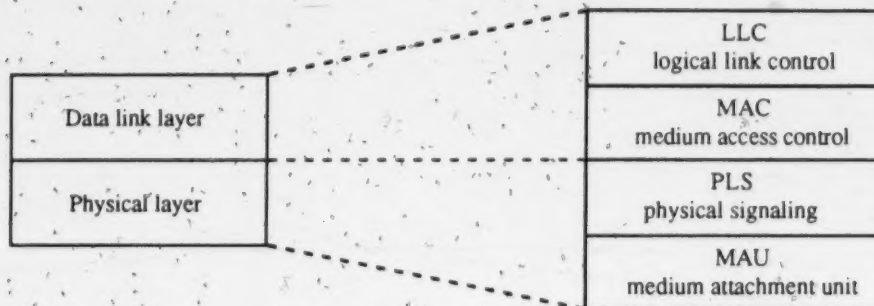


Figure 2.3 - CSMA/CD sublayers in the OSI model

CSMA/CD protocols act on the basis of three channel states: idle, busy or collide. The CSMA/CD standard specifies an enhanced physical layer to provide a channel status service. Figure 2.3 shows this layer split into medium attachment unit (MAU) and physical signaling (PLS) sublayers. The MAU sublayer performs bit transmission, the PLS sublayer monitors the channel and returns its status. The IEEE LAN standards (802.3 - 802.5) split the data link layer into medium access control (MAC) and logical link control (LLC) sublayers. The LLC sublayer specified by IEEE 802.2 is the same for all 3 LAN standards. LLC protocols ensure reliable, efficient and orderly delivery of network layer packets. To accomplish this, they embed packets into frame information fields and use cyclic redundancy checks (CRCs) to ensure frame correctness. Efficient delivery is provided through multiple buffers specified in the window size. The packet order is maintained by frame send and receive sequence numbers. In turn, this sublayer calls on the MAC sublayer to deliver each frame collision free to peer LLC entities.

The MAC sublayer implements protocols referred to as CSMA/CD. Upon receiving a frame, a MAC entity transmits it after possibly deferring to a current frame. If a collision occurs the frame is aborted and

retransmitted at a later time until an attempt is collision free or a limit is reached. The control of multiple frames is not implemented at the MAC sublayer. Each station has one sending and one receiving buffer. Frames are released when the MAC senses a successful transfer; this allows it to accept another LLC frame (if available) and commence transmission as soon as permissible. Peer (receiving) MACs usually receive frames correctly and pass them up to the peer (receiving) LLC. Other errors, such as inconsistent CRCs or lost frames are detected at the peer LLC. When this occurs a retransmission is requested at the LLC sublayer. The sending LLC then passes the frame to its MAC another time. This repetition is invisible to MACs, they transmit frames but do not interpret contents.

The MAC service specification produces a minimum frame length. Since frames are released when successfully terminated, it is imperative that stations hear collision(s) before completion. To derive the minimum frame length, define the channel length as L (time units) and channel status delay as S . Consider a station A that starts transmitting at time T . The physical layer of a station Z at the far end hears activity at $T + L$ and its MAC sublayer learns of the frame at $T + L + S$. Hence, Z may start transmitting at any time before $T + L + S$. A frame started at the last moment reaches the physical layer of A at $T + 2L + S$ and its MAC sublayer at $T + 2(L + S)$. Station A must be still transmitting so its absolute minimum frame length is $2(L + S)$. This quantity is called the synchronization period and represented by Δ^* . For centrally located stations the synchronization period is shorter, however CSMA/CD applies a homogeneous behavior. A margin of error δ added to this period yields the minimum frame length indicated here by Δ . In the Ethernet standard [ETHE80] the maximum synchronization period is 450 bits and the minimum frame length 512 bits. The final CSMA/CD specification of this thesis uses actual channel length and local clock errors to determine Δ^* and Δ . This provides a more stringent test of protocol correctness.

The sample protocols in the next section show that minimum frame length is related to virtual CSMA/CD slots. Within this thesis Δ is also called *slot length*. This period is sufficient for transmitting stations to know if other stations simultaneously access the channel. In a similar fashion, an idle period Δ ensures that other stations refrain from seizing an idle channel after a synchronization event. *In general Δ is used as the interval to synchronize network stations at the MAC sublayer.* This is the case for all global events in CSMA/CD such as packet transmissions, collisions and idle periods.

2.3. Two Sample CSMA/CD Protocols

This section describes two CSMA/CD protocols, one using an individual and the other a collective retransmission algorithm. CSMA/CD can be partitioned into subclasses based on these techniques. Both subclasses are modelled with *normal* and *retransmission* modes. The first mode is common, it statistically multiplexes new packets. This mode has no interstation synchronization other than deferring to current packets or collisions. This lack of co-ordination is referred to as uncontrolled behavior. Retransmission mode differentiates the two subclasses. A retransmission scheme in which stations act independently is also uncontrolled, whereas a co-ordinated retransmission is controlled. In this thesis, the name of the first (second) protocol subclass is *uncontrolled (semi-controlled) CSMA/CD*.

The initial CSMA/CD protocol is Ethernet as described in a paper by Metcalfe and Boggs [METC76]. Many attempts to improve Ethernet have spawned the CSMA/CD class. Ethernet is a member of the uncontrolled subclass in which only stations participating in collisions switch to retransmission mode, other stations remain in normal mode. The Ethernet retransmission algorithm is Binary Exponential Backoff, as described for Slotted ALOHA. In this algorithm a local collision counter (c) is reset when the station acquires a new packet and incremented for each collision. Following any collision, sending stations first emit a jamming signal. For its c^{th} packet collision, a station reserves 2^c virtual slots equal to the minimum frame length in which it can retransmit. Figure 2.4 shows the time line following the first collision of a packet. Stations randomly choose a slot and retransmit during that slot while deferring to current activity. Once the packet is transmitted without a collision, the station reverts to normal mode to acquire and transmit its next packet. After 10 collisions the number of slots is frozen at 1024. Following 16 collisions the MAC sublayer gives up and passes an error message to the LLC sublayer.

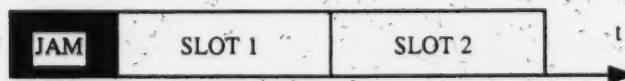


Figure 2.4 - Ethernet retransmission mode

CSMA/CD-DP (Dynamic Priority) illustrates the semi-controlled subclass in which all stations switch to retransmission mode after a collision. Retransmissions are synchronized on the basis of reserved slots, even for unslotted networks. An algorithm specifies which stations are required or forbidden to use each slot. When all reserved slots have elapsed, stations collectively return to normal mode. The DP slot access algorithm is simple, a slot is exclusively allocated to each station. Retransmission priority rotates by one station after each collision. Figure 2.5 shows the time line of an N station DP protocol network after a collision. It proceeds from normal or uncontrolled mode (U) through a jamming (J) and channel clearing (C) period. This is followed by N slots (S) before returning to normal mode. The privileged station is indicated within each slot.

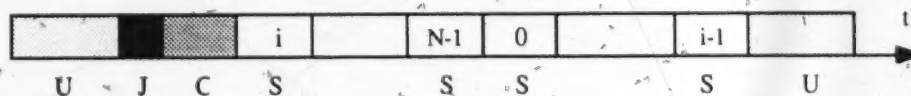


Figure 2.5 - Dynamic priority retransmission mode

The slots in figure 2.5 are not a fixed length as with slotted networks. Each slot represents a common acknowledgement that a station subset S^* has exclusive channel access for a period of time. With the DP protocol, the subsets S^* consist of a single station. The time line after collisions also includes a channel clearing delay to ensure jamming signals do not infringe on privileged stations in the first slot. The clearing delay is another example of a synchronization interval whose length is Δ and is viewed as slot 0. For other slots, their length must be sufficient to determine if a station(s) in S^* wishes to use the slot and if so, must allow the packet(s) enough time to finish (or collide). Their minimum length is Δ but can be extended if a station seizes the slot and requires a longer period to complete its transmission.

The fact that all stations switch to retransmission mode imposes an additional constraint: stations must maintain synchronized slots. This constraint does not exist for protocols such as Ethernet. To illustrate, assume an Ethernet station decides to retransmit in slot 10. If the station has a fast clock it may start its packet during slot 9. This is not a protocol error since the station was permitted to randomly choose slot 9. The point is: *uncontrolled protocols do not have global constraints that restrict channel*

access. For DP, a fast clock at station $(m+1) \bmod N$ may cause it to start retransmitting during the slot allocated to m . Several protocol errors could follow from such an incident. If station m also uses the slot an immediate retransmission mode collision occurs, an event forbidden by the protocol. If station m does not use the slot, other stations may disagree as to whether the packet was sent by station m or $(m+1) \bmod N$ based on when they hear the packet. Such disagreements desynchronize the slot count at different stations. This can result in future retransmission mode collisions if network stations are out of synchronization. There are two solutions to such problems, define a recovery procedure or try to prevent slot interpretation errors. Chapter 7 models a prevention technique that accounts for local clock variations.

The semi-controlled CSMA/CD subclass has advantages over the uncontrolled subclass. To start, semi-controlled protocols guarantee packet delivery in finite time. The DP protocol has a maximum packet delay on the order of N slots, other semi-controlled protocols are comparable. Ethernet cannot make delivery guarantees, there are only statistical success probabilities for each attempted transmission. A second semi-controlled feature is higher maximum throughput. Figure 2.5 shows that if most stations use their allocated slot, DP throughput can approach 100%. Slotted ALOHA has a corresponding maximum of 36.8%. A further point is that uncontrolled protocols are unstable. Instability is generally defined for an infinite station number with a finite offered load. In this situation, Slotted ALOHA maintains a maximum throughput of 36.8% for a finite time. Eventually, an unfortunate but inevitable run of collisions overloads the network. The protocol starts thrashing with this load and bogs down under an increasing number of waiting stations and a throughput that decreases to 0. To understand why, recall the features of Poisson packet arrivals. An offered load of λ packets/slot has probability $e^{-\lambda} \frac{\lambda^k}{k!}$ that k packets arrive during a slot. The probability of 1 packet is $\lambda e^{-\lambda}$ which represents the only situation for a successful transmission. This probability also equals network throughput and has a maximum of $e^{-\lambda}$ at $\lambda = 1$. If the offered load forces $\lambda e^{-\lambda}$ below the arrival rate of new packets, the network is in trouble. At this point, it receives new packets faster than it successfully transmits existing packets. The backlog increases which further decreases throughput and eventually the network grinds to a halt. Using Slotted ALOHA and an infinite station population, this instability occurs for any offered load above 0 [TANE88]. For a finite station number, protocol thrashing in an overloaded network is still present but not as serious.

To defend Ethernet and other uncontrolled protocols, one might argue that DP also has problems. An example is the mean delay in retransmission mode. This delay is $O(N\Delta)$ even for collisions involving only 2 packets. This is true but applies only to the simplest semi-controlled protocols. The Tree Collision Resolution (TCR) protocol studied later does not have this problem. Its average delay is $O(1)$ and worst case delay $O(\log_2 N)$ slots to resolve two station collisions. A sophisticated TCR version dynamically adapts to provide DP performance under heavy traffic loads.

These arguments demonstrate that the service specification of semi-controlled CSMA/CD protocols results in superior performance when compared to uncontrolled protocols. The price to be paid is a more complex protocol specification and implementation. In particular, it is difficult to specify slot timing constraints for the retransmission mode of semi-controlled protocols. The thesis goal is the modelling, implementation and verification of the semi-controlled CSMA/CD subclass. The DP protocol displays general subclass features, only minor changes are required for other members. In general, semi-controlled protocols have a dynamic number of retransmission slots with each slot allocated to a network station subset. The implementation detailed in this thesis is flexible enough to easily incorporate such variations. This is examined more thoroughly for the TCR protocol in the following chapters.

Chapter 3

Local Area Network Simulation Facility

This chapter introduces the reader to the Local Area Network Simulation Facility (LANSF). LANSF is a software package designed at the University of Alberta by P. Gburzynski and P. Rudnicki. It was originally created to investigate the performance of Medium Access Control (MAC) level protocols and has since evolved into a general modelling package for communication networks. A complete description of LANSF is found in [GBUR89a] or [GBUR89b].

MAC protocol specifications describe station behavior in an environment including timers, communication channels and clients. This environment sends *messages* such as passage of time, channel activity or client packet arrival. Stations respond to messages with internal actions and/or further message exchanges with the environment. LANSF stations are modelled as Communicating Finite State Machines (CFSM) in which state transitions are produced by messages from the environment. Chapter 1 listed general service properties that must be satisfied by LANSF implementations. FSM control states should anticipate all potential messages; otherwise the implementation is incomplete. Further, it is imperative that at least one awaited message arrives in finite time to avoid a deadlocked state. Stations must demonstrate progress to avoid livelocks. A sample MAC livelock is a packet with infinite delivery time because it continually collides when retransmitted. The stability property is not examined explicitly in this thesis, our simulation environment models ideal hardware.

The first section of this chapter presents a LANSF overview, it describes the simulating application and data file. The next section details the model of Communicating Finite State Machines used by LANSF stations. Section 3 shows how CFSMs are coded in the LANSF specification (implementation) language. The last section presents a specification of Ethernet as a warm up exercise to illustrate the existing CSMA/CD specification methodology.

3.1. An Overview of LANSF

LANSF is programmed using the C language in a UNIX[†] environment. Given a specification, LANSF produces a simulator to model networks using that protocol. These simulators operate as a collection of UNIX processes. Standard network operations are performed by processes executing predefined LANSF code. These generic processes produce the station environment and are automatically linked into simulators. They include the master process which creates and co-ordinates processes. Other generic processes implement tasks such as collecting statistics or simulating hardware. Processes simulating the station environment are called Activity Interpreters (AIs). Protocol specific (station) operations are performed by processes based on user defined specifications (C code). The three AIs used in our protocols are TIMER, CLIENT and LINK. All three provide immediate and future services, these correspond to immediate responses and future interrupts from the hardware layer. Immediate service requests are performed by calling functions reflecting their nature. A future service request (interrupt) is a call to function `wait_event(ai, event, state)`. Parameter `ai` is an integer specifying the appropriate AI and `event` provides additional event specific information. The last parameter indicates the new state following the event. Appendix A summarizes services used by CSMA/CD station processes described in this thesis.

The TIMER AI simulates a hardware clock. It returns the virtual time through variable `current_time` and informs stations when a specified delay has elapsed. This includes modelling local clock errors; a key physical limitation for MAC protocols. CLIENT simulates station clients generating packets. Its immediate service indicates whether a packet is available and if not, stations can request notification when one arrives. LINK simulates an ideal channel and its interface hardware. With LINK no packets are lost or damaged other than by collisions. LINK has immediate service requests to start, stop and abort packet transmissions or to start and stop jamming signals. It also returns the current channel status. Future service requests produce a wake-up event when the channel enters a specific state. This includes channel silence, the beginning or end of a packet transmission and the start of collisions. LINK does not simulate physical layer delays ($S = 0$), in LANSF the synchronization period Δ^* equals twice the channel length.

[†] UNIX is a registered trademark of AT&T Bell Laboratories.

The accuracy with which A/s reproduce the hardware layer is a key design feature of LANSF, particularly when considering implementation correctness. In this thesis, a generic state model is produced and verified for a subclass of CSMA/CD protocols in the LANSF simulated environment. This environment should reproduce the underlying hardware layer as closely as possible to minimize protocol modifications for a hardware environment. In this context LANSF is quite respectable, its built-in software functions have a close correspondence with physical layer hardware functions. As an example, the LANSF Ethernet transmitter implementation presented later in this chapter has a one-to-one state correspondence with the standard Ethernet specification [ETHE80].

Our CSMA/CD protocol implementations use the current LANSF methodology of stations specified by two processes, one for the receiver (logical input port), the other for the transmitter (logical output port). The two station processes are independent, no communication between them is required. Messages from a station process to the environment are based on LANSF functions. In contrast, messages from the environment to station processes are at the request of the latter as described previously. For example, when LINK simulates a packet transmission, it informs other stations of the packet arrival if the station has requested notification. In addition to exchanging messages with their environment, station processes require local variables to simulate memory in network stations. The user defines local variables in a LANSF include file; these are appended to the existing STATION structure created by LANSF for each network station.

LANSF simulations are customized by data files of network parameters. These include the number of stations and ports per station. CSMA/CD stations each have one port aliased as BUS. Data files state the number and type of network channels (links in LANSF). They also specify port/link connections and station locations. Following this, input files describe network (client) traffic which includes distribution, size and frequency of packets. Our traffic was uniform, each station sent and received packets with equal probability and with complete interstation communication. This addresses the problem of partial network locks discussed in chapter 1. The last portion of data files contains protocol specific parameters. The Ethernet specification uses this to describe features contained in its standard [ETHE80].

3.2. Communicating Finite State Machines

Consider now the LANSF model of protocol entities which are Communicating Finite State Machines. A CFSM is defined as a FSM capable of exchanging messages with other CFSMs and/or some environment. When a CFSM communicates with a second CFSM the message is transmitted through their common environment, hence all communications occur between CFSMs and the environment. Simulated CFSMs operate in discrete virtual time. An immediate service request occurs instantaneously which approximates the physical situation of high speed CPUs in MAC entities. Future services produce an interrupt at some time equal to or later than the current time T .

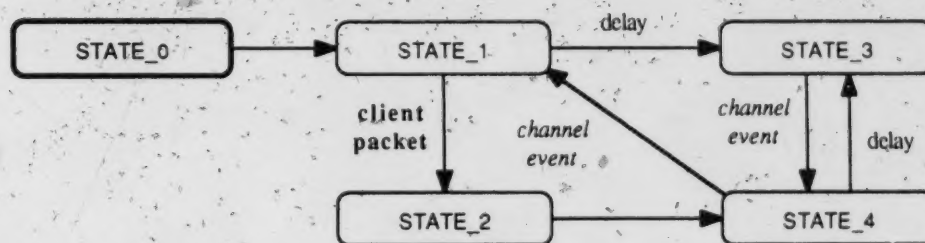


Figure 3.1 - A Communicating Finite State Machine.

Figure 3.1 illustrates a sample CFSM state/transition diagram. States are depicted with rounded rectangles and transitions by directed arcs. LANSF CFSMs have one initial state **STATE_0** indicated by a bold outline. There are no final states since correct protocols never terminate. Each control state has *actions* and *transitions*. Actions are performed immediately upon entering a state; they are not shown in the figure. After performing state actions, CFSMs inform the environment of anticipated (transition) events and wait for one to occur. Virtual time flows only while CFSMs wait for transition events; no time elapses if an event is available immediately. Otherwise, a transition is fired by the first temporal event. If multiple events occur at the same discrete time, one is randomly chosen by the environment. Unlabeled transitions such as that from **STATE_0** are immediate, no event is required. Labelled arcs require an event, transitions in **bold** are local events usually provided by **CLIENT**. Those in *italic* are events delivered by **LINK**. Transitions labelled with plain text correspond to the passage of virtual time, **TIMER** provides these events.

3.3. The LANSF Specification Language

This section reviews LANSF specifications of protocol entities. Station behavior is simulated by processes executing user C code. Hence in LANSF, *protocol specifications* and *implementations* are identical. This low level specification language has strong and weak points. The strong point is that it closely maps hardware details and implementations are easily ported to real networks. As stated earlier, LANSF attempts to provide an *AI* service corresponding to each physical layer service. The weak point is a lack of specification abstraction. Formal languages specify minimal information to describe protocols. They ignore implementation details which permits users to easily reason about specification properties. In detailed low level specifications such reasoning is difficult, it resembles not seeing the forest for the trees. A timely question to ask is: *Is it possible to express a high level specification of protocol properties with LANSF?* The answer is YES, this thesis uses LANSF observers for that purpose as will be demonstrated.

LANSF station processes are created during initialization with calls to `new_process (code, version)`. In CSMA/CD the two processes per station typically utilize code from functions named `receiver` and `transmitter`. Station process code has the generic format of a single function, a skeletal version is shown below.

```
station_process () {  
    switch (the_action) {  
        case INITIALIZE:  
            init_actions;  
            init_transitions;  
            return;  
        case STATE_1:  
            state_1_actions;  
            state_1_transitions;  
            return;  
        ...  
        case STATE_N:  
            state_n_actions;  
            state_n_transitions;  
            return;  
    }  
}
```

The case constants of the switch statement represent control states. A call to station_process branches to the state indicated by the_action where it performs state actions and specifies transition events. LANSF defines constant INITIALIZE, new processes are initially awakened into this control state. For efficiency station processes are awakened only when transition events occur. After returning from its call, a station process is put to sleep. Let us now examine this specification language in action with a sample protocol.

3.4. An Ethernet Specification

The Ethernet specification demonstrates some current LANSF methodology used to implement CSMA/CD protocols. An early choice is the simulation time unit. With CSMA/CD, it generally equals the time to insert a single bit onto the channel. For 10 MBit/sec Ethernet, this unit equals 0.1 μ sec. Virtual time is represented by type TIME which is an extended integer with a length of 1 to 5 integers. The specifications here use a single integer so type TIME equals int. This is not general LANSF methodology but is used here to simplify specifications. With this choice the simulation time must not exceed 2^{31} bits (~200 sec).

The Ethernet specification is expressed as two non-interacting processes per station called receiver and transmitter. The receiver process code is generic to all CSMA/CD protocols and is the following:

```
receiver () {
    switch (the_action) {
        case INITIALIZE:
        case WAIT_FOR_PACKET:
            wait_event (BUS, END_MY_PACKET, PACKET_RECEIVED);
            return;
        case PACKET_RECEIVED:
            accept_packet (the_packet, the_port);
            skip_and_continue_at (WAIT_FOR_PACKET);
    }
}
```

This process is awakened into state INITIALIZE at virtual time 0. From there, control proceeds to state WAIT_FOR_PACKET according to C switch statement semantics. The latter state requests notification when the end of a packet addressed to its station occurs. This produces a transition to PACKET_RECEIVED where an immediate service request (accept_packet) to CLIENT enters the packet into simulation statistics. A TIMER request delays for one unit before returning to WAIT_FOR_PACKET. This delay is needed in LANSF so that state WAIT_FOR_PACKET does not repeatedly process the same packet. Figure 3.2 summarizes the behavior of receiver processes.

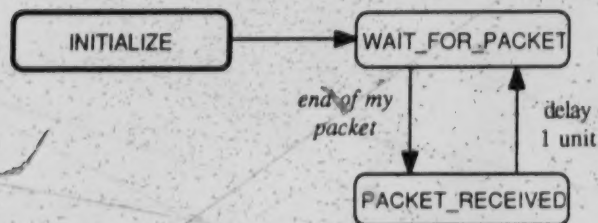


Figure 3.2 - CSMA/CD receiver states/transitions

Figure 3.3 shows the transmitter process control state diagram based on a version in [GBUR89b]. It has essentially a one-to-one state mapping with the Ethernet standard FrameTransmitter process except that it does not deal with excessive collisions. This process uses one station variable called collision_counter appended to the generic STATION structure. Appendix B presents the complete transmitter specification.

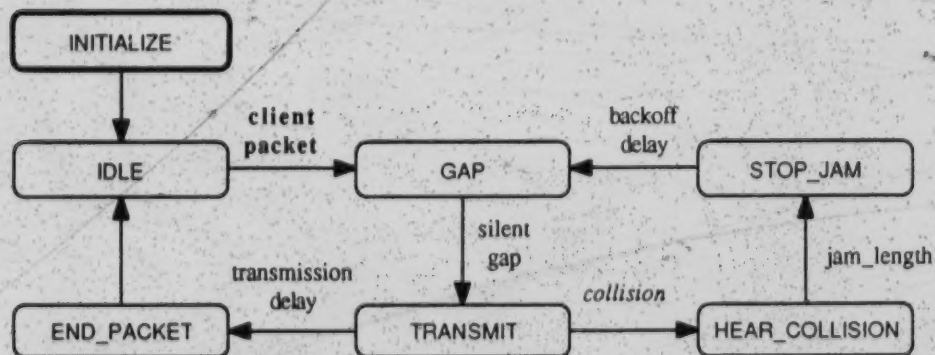


Figure 3.3 - Ethernet transmitter states/transitions

Control enters INITIALIZE at virtual time 0 and proceeds immediately to IDLE. Station transmitters leave IDLE only when they receive a client packet. This demonstrates the uncontrolled nature of Ethernet: stations without packets do not monitor the channel. Stations receiving a packet enter state GAP, this state first defers to current channel activity and then enforces an inter-packet gap. Upon leaving GAP control transfers to TRANSMIT which initiates transmission. This state sets wake-up events for two possibilities, a successful transmission or a collision. The first is represented by a delay equal to the transmission period. The second is a LINK request for notification if a collision occurs. A successful transmission transfers control to END_PACKET which terminates the packet. Following this is an immediate return to IDLE.

A collision transfers control to HEAR_COLLISION. In this state, transmitting stations abort their packets and emit jamming signals. Jamming signals are additional noise to ensure that collisions propagate to all points on the channel with sufficient intensity to be recognized. After the jamming period of a few bytes, control switches to STOP_JAM which terminates the signal and calculates a backoff delay. For Ethernet, this delay equals a random number of slots (512 bits) in the range $[0, 2^c - 1]$ where c is the collision count. After this delay, control re-enters state GAP. The station is then in a similar situation as when it first received the packet. It defers, enforces an inter-packet gap and starts transmission. The only difference results from the backoff delay calculation if another collision occurs.

This implementation illustrates features of uncontrolled CSMA/CD protocols. The Carrier Sense (CS) is implemented by state GAP which defers to current traffic. Collision detection is included in state TRANSMIT. The uncontrolled nature is due to local determination of packet retransmission time found in STOP_JAM. It is worth repeating that non-transmitting stations do not sense the channel; if a collision occurs they remain in normal mode. These features contrast those of a semi-controlled CSMA/CD protocol (Tree Collision Resolution) described in the next chapter.

Chapter 4

Tree Collision Resolution

This chapter discusses the initial protocol specification verified with LANSF observers. This specification was independently written and presented to the thesis author for conformance testing. There were no known errors prior to conformance testing. A visual code check by the two specification authors revealed no errors and the network throughput based on simulation results was within expected bounds. Conformance testing then revealed an error and an ambiguity that were not discovered by informal correctness tests. A detailed model of the second specification derived after conformance testing is presented in chapters 7 and 8.

The case study, Tree Collision Resolution or TCR, is a semi-controlled CSMA/CD protocol. Its normal mode is the same as Ethernet. During retransmission mode, TCR implements globally synchronized slots. Stations obtain channel access privilege during slots on the basis of a Tree Collision Resolution algorithm. The original work on MAC protocols using TCR is due to Capetanakis [CAPE79]. In this paper, he proposed a new class of algorithms for slotted networks and examined their performance. His abstract reads:

Previous accessing techniques suffer from long message delays, low throughput and/or congestion instabilities. A new class of high-speed, high-throughput, stable, multiaccessing algorithms is presented. ... It is also shown that, under heavy traffic, the optimally controlled tree algorithm adaptively changes to the conventional time-division multiple access protocol.

For slotted networks it is easily shown that TCR protocols have a higher maximum throughput than uncontrolled protocols. Capetanakis states that an adaptive TCR protocol behaves as Dynamic Priority (DP) for high traffic loads. Consider the maximum DP throughput for an N station network. If collisions occur during each uncontrolled slot, DP sequences have a normal mode slot followed by N retransmission slots. The maximum throughput of $\frac{N}{N+1}$ packets/slot occurs when all stations are backlogged. This is the ideal situation, real networks do not perform this well. The two protocols described by Capetanakis in

[CAPE79] are *static binary tree* and *optimum dynamic tree*. Their maximum throughputs are 0.347 and 0.430 packets/slot respectively, the latter being 17% above Slotted ALOHA. A second point is that TCR (and other semi-controlled) protocols are stable. Capetanakis proves that all packet delay moments for TCR are finite. Stable protocols do not thrash when overloaded; they indefinitely serve any offered load below their maximum throughput. In fact, the DP protocol only reaches maximum throughput when all stations are backlogged. The general result about bounded TCR moments demonstrates the third property, that TCR has a bounded packet delay which is the first moment. This is also a general result with semi-controlled protocols, the maximum packet delay of this subclass is $O(N\lambda)$.

The model used by Capetanakis assumed an ideal slotted channel, his interest was protocol performance. Further research with slotted networks has been performed to study TCR robustness in the presence of channel errors. Early work considered uniform errors common to all protocol entities. Recent work by Suda et al. [SUDA90] considers TCR robustness to non-uniform errors. Their conclusion is that TCR is robust and compensates for these error types. The slotted model is useful because it simplifies research of performance and robustness on TCR and similar protocols. Its abstraction removes the problem of implementing synchronous slots.

For most networks, slotted models are not realistic situations. Local Area Networks (LANs) are usually asynchronous and "slots" are created only when needed. Therefore, is the existing robustness research applicable to CSMA/CD? The answer is a qualified yes; results proven for slotted networks can be partially transferred to unslotted (CSMA/CD) networks. It must be shown that CSMA/CD slots are consistently interpreted by network stations, this is the slotted model abstraction. This proof can be tricky, for example, consider a situation where a sending station views a successful packet transfer and another station erroneously hears a collision. If the second station detects the "collision" towards the end of the packet, it may jam the channel such that the jamming signal reaches the sending station after terminating its packet. It is not obvious that both stations then sense the collision in the same slot so the proof of synchronous CSMA/CD slots in this situation is by no means trivial. Such a proof is beyond the scope of this thesis; synchronous slots are implemented here for an ideal channel.

Previous research does not demonstrate how to implement controlled mode slots for semi-controlled CSMA/CD protocols. This chapter examines the initial attempt at a LANSF specification for TCR which incorporates this feature. The full specification is found in Appendix C, it is taken from [GBUR89b] with minor changes. This chapter starts with an overview of semi-controlled protocols in a slotted network, using the DP protocol as an example. It then examines how TCR operates with this type of network. In the last section, the initial LANSF specification of CSMA/CD-TCR is presented.

4.1. Slotted Semi-Controlled Protocols

In chapter 2, an uncontrolled slotted protocol (Slotted ALOHA) and a semi-controlled unslotted protocol (CSMA/CD-DP) were presented. Protocols such as DP or TCR implemented on slotted networks are called semi-controlled slotted protocols. A sample time line for slotted-DP is shown in figure 4.1.

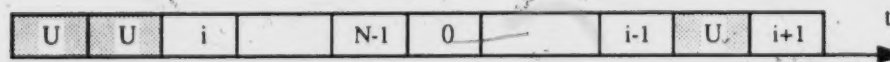


Figure 4.1 - A slotted DP time line

This time line is similar to figure 2.5 for CSMA/CD-DP. There are also differences, the main one is that normal mode also has slots labelled "U". These normal mode slots permit channel access to all stations but enforce synchronous transmissions. Other differences include the lack of channel jamming and clearing periods after collisions. The slotted model dictates that stations learn the channel status (idle, 1 packet, collision) before slots terminate. For semi-controlled protocols, uncontrolled collisions switch the network into controlled mode. Controlled mode is characterized by restricted channel access; each slot has a privileged station subset S^* .

For slotted semi-controlled protocols, each Collision Resolution Interval (CRI) consists of an uncontrolled slot plus any following controlled slots. The name CRI is taken from a TCR robustness paper by Suda et al. [SUDA90]. The shortest CRI is an uncontrolled slot that is idle or has a single packet. If an uncontrolled slot suffers a collision, its CRI includes controlled slots used to resolve the initial (uncontrolled) collision and subsequent (controlled) collisions. For DP, CRIs are always 1 slot when there

is no uncontrolled collision or $N+1$ slots (uncontrolled collision) as collisions do not occur in controlled mode. Slotted semi-controlled protocols have a common normal mode behavior, their differences are due to allocation of controlled slots. In general these protocols have a dynamic controlled slot algorithm, TCR is one such example.

4.2. Slotted Tree Collision Resolution

Consider now a slotted protocol using Tree Collision Resolution for controlled mode. The TCR algorithm is as follows: If an uncontrolled collision occurs, divide network stations into two subsets of comparable size and reserve a controlled slot for each subset. The best scenario occurs when uncontrolled collisions involve exactly two stations, one from each subset. In this case the two initial reserved slots are adequate to resolve the collision. However, if one or both controlled slots suffer a collision then further action is required. The station subset for which the collision occurs is subdivided into two smaller subsets and an additional slot is reserved for each subset. This process is performed recursively until the subsets are such that at most one station attempts to access each controlled slot. The worst case occurs when a subset must be reduced to a single station before meeting this condition.

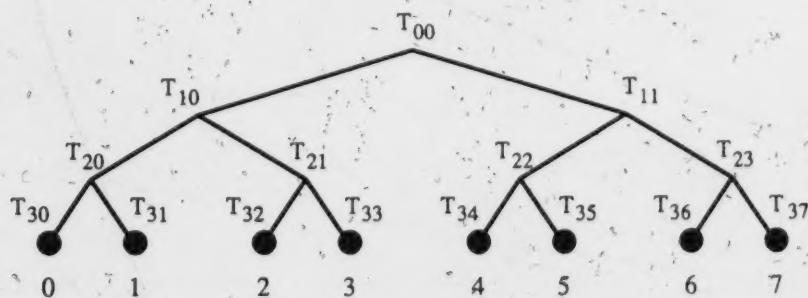


Figure 4.2 - A sample TCR virtual binary tree

With TCR, station subsets are determined according to a virtual network balanced binary search tree. Figure 4.2 shows a tree for a sample network of 8 stations. Subtree T_{ij} is defined as the j^{th} subtree from the left in the i^{th} level and the entire tree is equivalent to T_{00} . Tree leaves (labeled $0 \dots 7$ and T_{3j} for $j = 0 \dots 7$)

represent stations. TCR uses subtrees T_{ij} as privileged subsets during controlled mode. To maintain consistent notation, the privileged subset (subtree) is changed from S^* to T^* for this protocol.

The TCR algorithm presented here is from [SUDA90]. This version is chosen because our CSMA-CD implementation resembles it more than that of Capetanakis [CAPE79]. Capetanakis applies TCR to slot pairs. Our unslotted algorithm and the [SUDA90] slotted algorithm apply TCR to individual slots. In slotted TCR, uncontrolled mode has privileged subtree T_{00} and remains active until a collision occurs. Following uncontrolled collisions two controlled slots are reserved, one for T_{10} and the other for T_{11} ; the children of the previous privileged subtree T_{00} . If controlled mode collisions occur, two more slots are reserved for the children of the current T^* . Hence if T_{xy} is privileged during a controlled collision, slots are reserved for $T_{x+1,2y}$ and $T_{x+1,2y+1}$. Slots are reserved in a left to right prefix order of their privileged subtree. As a consequence newly reserved slots precede existing slots and the reservation schedule is viewed as a stack. Controlled mode terminates when all reserved slots have elapsed.

For TCR a Collision Resolution Interval is called a tournament, a name suggested by the tree resolution of contestant positions in tournaments. Our implementation places an additional restriction on controlled mode. Only stations attempting to transmit during the tournament uncontrolled slot can participate. This corresponds to stations with packets before the tournament commences, i.e. stations that register in time. This restriction is a convention and not needed to obtain the TCR service properties discussed shortly. Stations with packets when tournaments start are called tournament *participants*. Slotted TCR is defined as an infinite loop of the following steps. Step 1 represents uncontrolled mode and is the start of new tournaments. Steps 2 and 3 describe controlled mode.

1. Execute an uncontrolled mode slot ($T^* = T_{00}$). If no collision occurs repeat this step, otherwise collision participants register themselves for controlled mode and all stations proceed to step 2.
2. Reserve two slots for the subtrees (children) of the current privileged subtree. Proceed to step 3.
3. If the reservation schedule is empty return to step 1, otherwise execute the first scheduled (controlled) slot. If a collision occurs go to step 2, otherwise repeat this step.

A sample tournament can be demonstrated for the network of figure 4.2. Assume that stations 0, 2, 6 and 7 have packets at the start of a tournament. The resulting tournament is shown in figure 4.3.

Slot	1	2	3	4	5	6	7	8	9
Privileged subtree	T_{00}	T_{10}	T_{20}	T_{21}	T_{11}	T_{22}	T_{23}	T_{36}	T_{37}
Transmitting stations	0 6 2 7	0 2	 0	 2	6 7	 	6 7	6	7

Figure 4.3 - A slotted TCR tournament

The first (uncontrolled) slot produces a collision between the 4 competing stations. These stations register as participants, others cannot use controlled tournament slots. Initially, there are two slots with privileged subtrees T_{10} and T_{11} respectively. Stations 0 and 2 transmit during the first slot and produce a collision which reserves slots for T_{20} and T_{21} . Both new slots execute without collisions as the first (second) is claimed by station 0 (2). Afterwards, packets from stations 6 and 7 collide during the T_{11} slot, this reserves slots with T_{22} (T_{23}) privileged. This first slot is unused, the second produces another collision. The final two slots are for subtrees of T_{23} and correspond to stations 6 and 7. Their execution terminates the tournament as both slots cannot suffer collisions. A new tournament then starts with the return to algorithmic step 1. Note that all tournament participants have successfully transmitted.

Tournaments can be depicted with binary search trees as explained in [SUDA90]. The tournament tree for the previous example is shown in figure 4.4. A comparison shows that this tree is similar to the network tree of figure 4.2. The common nodes are in the same location for both trees. However, the latter tree contains only subtrees visited during the tournament. Nodes are labelled by the privileged subtree of the corresponding slot, the root node represents the uncontrolled slot with $T^* = T_{00}$. Each collision appends two children to the node depicting the corresponding slot. Internal nodes represent slots in which collisions occurred, leaves correspond to slots without a collision. The slot order is determined by a prefix left to right traversal. In the recursive TCR algorithm, tournaments can contain subtournaments. A subtournament is defined as a slot in which a controlled mode collision occurs and successive slots used to resolve that collision. To illustrate in figure 4.4, the subtree rooted at T_{10} corresponds to a subtournament which resolves the contention between stations 0 and 2.

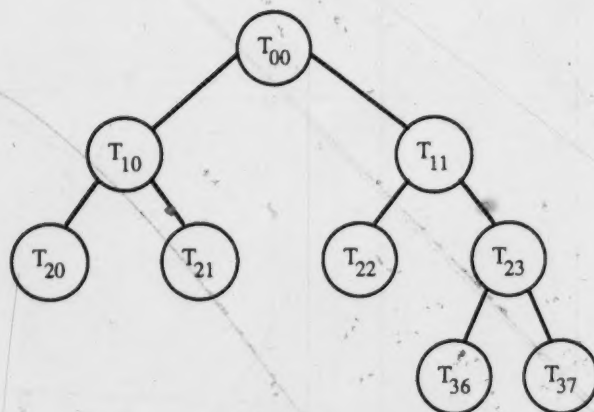


Figure 4.4 - A tournament tree for slotted TCR

Tournament trees help to reason about protocol properties. It was claimed that the maximum packet delay for this protocol subclass is $O(N\Delta)$. To demonstrate for TCR, consider the maximum tournament length. A tournament tree has at most N leaves which occurs if each station is visited individually. Binary trees with N leaves have $N - 1$ interior nodes so an N station tournament has at most $2N - 1$ slots. The maximum packet delay is less than 2 complete tournaments or $4N$ slots. A second important protocol property is the maximum number of consecutive idle slots. This defines the period over which stations maintain synchronization without communication. To derive this quantity first note that deeper nodes are only reached after collisions. Two sibling nodes cannot both be idle as their parent suffered a collision. Hence, two or more consecutive idle nodes occur only when the traversal returns to the root. The longest such path equals the number of tree levels excluding the root and leaves or $\log_2 N - 1$. This path occurs when the first two stations, 0 and 1, compete for the channel and produce $\log_2 N$ consecutive collisions preceding successful transmissions. The $\log_2 N - 1$ right sibling nodes are then traversed sequentially. TCR produces less difficult synchronization problems than DP which permits up to $N - 2$ consecutive idle slots.

4.3. LANSF Specification of CSMA/CD-TCR

The initial CSMA/CD-TCR specification is listed in Appendix C. Its retransmission algorithm is intended to be conceptually identical to slotted TCR. CSMA/CD produces implementation differences during controlled mode due to the unslotted network. A general service specification of semi-controlled CSMA/CD protocols is to maintain synchronous controlled slots which differ from those of slotted networks. To start, CSMA/CD does not assume negligible channel length. This length and physical signalling delay are considered by the synchronization period Δ^* described in chapter 2. With perfect clocks Δ^* defines the minimum slot length. The maximum slot length has no theoretical limit but is determined by performance criteria. If a station seizes a slot, other stations wait until it has finished so transmitting stations are responsible for respecting packet length limits. CSMA/CD networks also treat collisions differently, they attempt to silence the channel as soon as possible.

Let us state additional service properties that the specification of Appendix C intends to satisfy. The uncontrolled mode properties listed previously are straightforward and not subjected to conformance testing.

The claimed properties of controlled-mode are as follows:

1. Stations transmit only when they are within the privileged subtree
2. Any station not participating in an uncontrolled (controlled) collision will not transmit during the subsequent tournament (sub-tournament).
3. All stations participating in an uncontrolled (controlled) collision successfully transmit during the subsequent tournament (sub-tournament).

The process of verifying these properties is considered in chapter 6. For now consider the initial CSMA/CD-TCR implementation. Figure 4.5 shows the transmitter normal mode states. As with Ethernet, stations start in INITIALIZE and proceed immediately to IDLE. Stations wait in this state until it is time to transmit or bus activity is sensed. Upon entering IDLE, an initial test prevents a station with a full buffer from acquiring another packet. This suggests a potential problem with respect to property 3 listed above. Is it possible for stations to be excluded from tournaments and still have buffered packets when returning to IDLE? This will be determined in Chapter 6. Continuing the semantics of IDLE,

- stations that have or acquire a packet upon entry prepare to transmit. Before starting a transmission, stations check for channel activity and if present proceed to CHANNEL_BUSY until the channel is idle. Otherwise, a timer is set to enforce an inter-packet gap provided one has not yet occurred. The function of Ethernet state GAP has now been incorporated into IDLE. After the gap, stations transfer to TRANSMIT. In addition to starting the transmission, this state sets a local variable to differentiate transmitting stations:

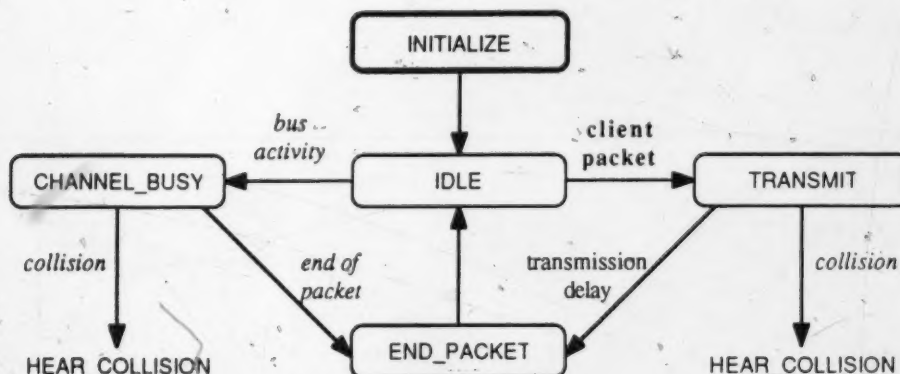


Figure 4.5 - CSMA/CD-TCR normal mode

With semi-controlled CSMA/CD protocols, all stations must know when collisions occur as they collectively switch to controlled mode. Another difference of IDLE compared to Ethernet is that stations hearing channel activity transfer to CHANNEL_BUSY. In this state they wait for a successful packet termination or a collision; both events synchronize stations. During controlled mode stations must know when packets terminate as this event signals the end of a slot. For uncontrolled mode, the end of a successful transfer informs stations that they can freely compete for bus access from state IDLE.

The controlled mode states of this protocol are shown in figure 4.6. Stations enter this mode at state HEAR_COLLISION upon hearing a collision. In this state, transmitting stations emit a jamming signal and proceed to STOP_JAM. The latter state terminates the signal and enforces a channel clearing delay. In state HEAR_COLLISION, non-transmitting stations set a delay to wait through the jamming signals of transmitting stations and the channel clearing period. All stations enter state COLLISION_GONE at time $\text{jam\_length} + \text{collision\_delay}$ after first hearing the collision.

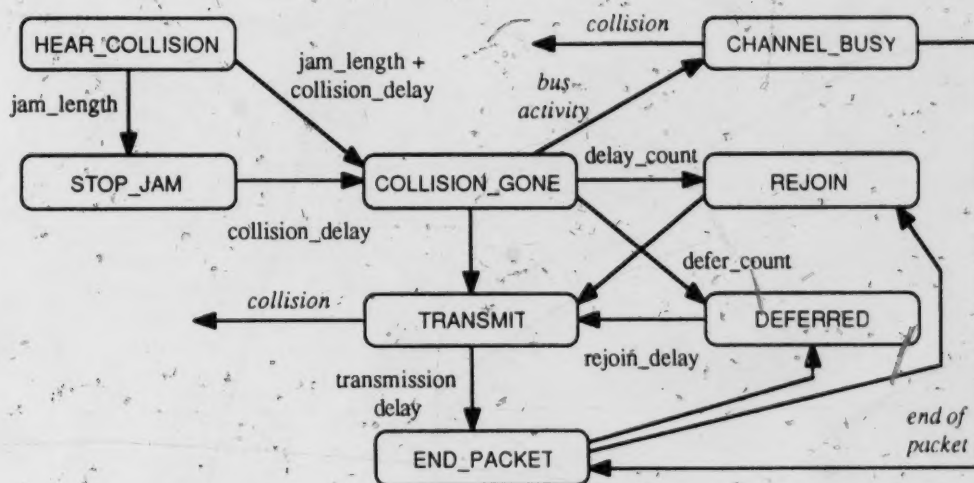


Figure 4.6 - CSMA/CD-TCR retransmission mode

The transition to COLLISION_GONE represents the start of the first slot after a collision. During controlled mode, stations maintain tournament information using a boolean and two integer variables. The boolean flag called `tournament_in_progress` is set when entering COLLISION_GONE and reset by state IDLE. The integer variables are `defer_count` and `delay_count`. The first indicates the number of controlled slots remaining, the second how many slots until the station is within the privileged subtree. The latter variable is maintained only by tournament participants.

After setting tournament variables in COLLISION_GONE, stations test if they are permitted to use the first slot. To do so, they must have a buffered packet (`transfer_pending`) and be within the privileged subtree (`left_subtree`). Packet possession is equivalent to being a participant as stations are forbidden to buffer packets during tournaments. A station meeting both criteria proceeds to state TRANSMIT and immediately commences its transmission. Stations that fail either criterion set a timer for `delay_count` (`defer_count`) slots if they have (don't have) a packet. Subsequent events depend on what happens during the slot. If another collision occurs, stations re-synchronize and reset tournament variables by proceeding through the same state loop as for uncontrolled collisions. If the slot is idle but there are additional participants, one or more stations eventually times out and proceeds to REJOIN. In this state, stations delay for a short period before continuing at TRANSMIT. This delay compensates for clock errors, without

it stations could de-synchronize. For example, if station J (K) has a fast (slow) clock and both stations wish to use the same slot, a packet from station J can start and reach K before the latter timer elapses. Station K incorrectly assumes the packet was intended for the previous slot and defers to station J . The delay in REJOIN insures that all stations correctly realize the previous slot termination. A similar technique is used when stations collectively return to uncontrolled mode (state IDLE). This situation has a branch to state DEFERRED when a station perceives a tournament as finished. After a short delay, the station continues at IDLE. This delay insures that slow stations do not misinterpret an uncontrolled mode packet from a fast station as a packet intended for the last controlled slot.

The last case for a controlled slot is being used by a single station. The sending station transmits its packet and branches to END_PACKET. After releasing its packet, the station is no longer a participant so it sets a timer for an inter-packet gap plus defer_count slots at which time it returns to normal mode via state DEFERRED. Listening stations also branch to END_PACKET when they hear the end of packet. In a similar manner, they set timers for a gap plus defer_count or delay_count slots with the intent of returning to normal mode or using an awaited slot respectively.

This specification provides a prototype for semi-controlled CSMA/CD protocols. It claims to enforce synchronous slots via the delay mechanism that transfers control to DEFERRED and REJOIN. In addition, it attempts to satisfy the service specifications listed earlier in this section. Other semi-controlled protocols can be specified simply by changing the calculation of variables defer_count and delay_count. So now we pose the question: Does this specification accomplish its intended purpose? To answer this question, the next two chapters discuss software observers used to formalize the service specifications and perform implementation conformance tests.

Chapter 5

Implementation Observers

As stated previously, the primary goal of this thesis is to generate a verified CSMA/CD-TCR implementation. The process of verifying implementation conformance to a standard is *conformance testing* and the conclusion is called a *verdict*. Verification has two components, *syntactic* and *semantic*. Validation or syntactic verification insures correctness of general properties. The existing LANSF methodology provides support for validation. Throughput can indicate a negative verdict regarding conformance to general properties such as absence of locks. Further, LANSF functions contain self-checking code. For example, a run time error occurs if a buffer being filled already contains a packet. Self-checking code helps to detect general programming errors. With these features, implementation non-conformance to general properties should be detected if an erroneous state is explored during simulation.

In contrast, traditional LANSF protocol development provides little support for semantic verification of implementations. There are no existing LANSF (mechanical) techniques to perform implementation conformance tests to their protocol specific properties. It is possible to explore semantically incorrect states during simulation and be unaware of this fact. This problem of verifying semantic correctness starts at the beginning of the LANSF protocol development cycle. The LANSF specification and implementation languages are the same; informal service specifications are directly coded into implementations. There is no high level specification language to reason about protocol properties prior to implementation. The difficulty of reasoning with low level languages is well known, this was a factor driving the development of FDTs. However, reasoning based on implementations is all that is currently available to LANSF protocol developers prior to simulation.

To correct this deficiency in LANSF MAC protocols, existing OSI-FDT research suggests two independent paths. The first is to develop a high level specification language applicable to MAC protocols. This is beyond the scope of this thesis and is not considered further. A second path is to generate

implementation verification tools. This path is partially dependent on the first; if a formal specification language exists, implementation verification tools are designed on the basis of this language. *To mechanically perform semantic verification without a formal language, service specifications must be encoded into tools.* Then, implementation conformance to this standard can be automatically performed. The tool to be used for formal service specification and mechanical verification is called a LANSF observer. Each observer will be programmed (specified) to monitor implementation conformance to a single TCR service property given in the previous chapter.

Before examining LANSF observers or specific tests they perform, it is useful to review existing research related to verifying protocol implementations. The first section examines existing conformance testing architectures and results, which is primarily in the area of FDT specifications. Section 2 examines unique observer features required for CSMA/CD protocols. In the last section, the LANSF observer model and specification language are discussed.

5.1. Implementation Conformance Testing

Most of the existing conformance testing research relates to physical networks and the majority to implementations of OSI-FDT specifications. A recent article on OSI protocol conformance testing is by Linn [LINN90]. He lists three techniques for conformance testing a single layer which are the local, distributed and coordinated methods. The coordinated method is conceptually similar but more difficult than the distributed method and is not considered. The local test method corresponds to a *local observer* and the distributed method to a *global observer*. Both observer types can operate in *active* or *passive* mode. Passive observers only monitor implementations, an underlying assumption is the existence of at least two protocol entities to communicate. Active observers exchange messages with the implementation under test (IUT). In this case the observer and implementation comprise the "network". The existing literature on OSI-FDT observers primarily considers active observers.

In the local method, the lower and upper interfaces of the IUT (layer n) are both exposed. These interfaces are used as *points of control and observation* (PCOs) by active observers or *points of observation*

by passive observers. Linn does not consider this architecture with passive observers. An upper (lower) tester is connected to the upper (lower) interface of the layer as shown in figure 5.1.

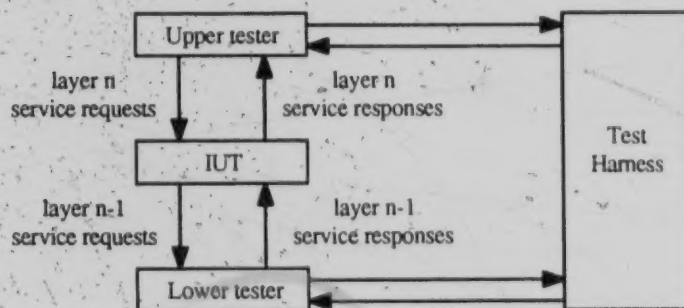


Figure 5.1 - Local conformance test system architecture

As an active observer, system test harnesses use PCOs to control IUT inputs and observe outputs. Layer n conformance is tested by exchanging layer n service primitives at the upper interface and noting layer $n-1$ primitives at the lower interface. The test harness coordinates the interfaces and logs essential information. This method *tests implementation conformance to the protocol specification*. This includes static features such as data unit encoding and dynamic features which include event sequencing. A positive verdict means that layer n is correctly implemented in terms of layer $n-1$ services. The local test method does not verify global service specifications. This logical extension is derived from an a priori proof that the protocol specification meets service specifications. The local method is applicable when there is a protocol specification to use as a standard, this includes FDT specified protocols but excludes MAC protocols.

The distributed method does not assume an exposed interface (PCO) below the IUT, however its upper interface is still exposed (see figure 5.2). The upper tester imitates a layer $n+1$ entity within the same unit that requests IUT services. With this method access to the lower interface is through the communication medium. The lower tester and implementation are located in two different units and the lower tester is a layer n peer to the IUT. These peers communicate through the physical medium by requesting layer $n-1$ services. A test harness interconnects the upper and lower testers and collects information from both peers.

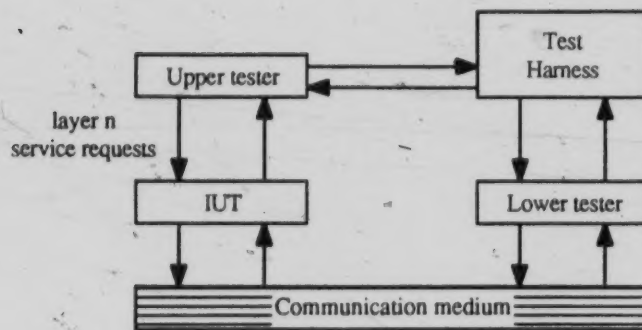


Figure 5.2 - Distributed conformance test system architecture

Global observers (distributed test systems) check multi-entity, or global properties. This test system as described in [LINN90] is an active observer because upper testers request IUT services. This distributed method is also poorly suited to MAC protocols. The combination of IUT and its peer (lower tester) define the "complete" network which is reasonable for FDT specified protocols based on two entities. With these protocols, the global state is a product of two local states $\phi_0\phi_1$, one for the lower tester and one for the implementation. The test harness knows the lower tester state ϕ_0 and can determine the expected IUT state ϕ_1 from interactions with testers. Since two local states define the global state, the harness knows the anticipated IUT behavior at all times and can render a conformance verdict. The situation with multi-entity MAC protocols is more complex. Here a test system for an N-entity network requires N upper testers and 1 lower tester, the latter being a peer to MAC entities. In monitoring this network, the harness must deal with the recurring MAC problem of state space explosion. In addition, the harness must interpret complex channel timing events. This feature is doubly complex because entities are physically separated from one another and the test harness is separated from the testers. In summary, there is too much information in a physical MAC network to be monitored by a distributed test system created for the FDT model.

These two testing methods are complementary for FDT protocols as they are essentially orthogonal. The local method tests IUT correctness with respect to vertical interfaces (figure 1.1), the distributed method considers horizontal interfaces. The local method performs single entity tests that exclude potential errors in lower layers. It is suited to initial testing within a friendly environment. The distributed method studies the implementation in an actual environment which produces hardware errors and possible software errors in

lower layers. Distributed testing also considers the effects of physical separation between entities. This complicates the test harness as it must consider transmission delays. For another comparison of local and global observers oriented towards FDT specifications, the reader is referred to [DSSO86]. In this paper, the authors examine combining local and global observers as well as comparing the error detecting capability of test architectures. They show that global observers have a greater error detecting capability.

A paper describing a physical MAC protocol run time observation is by Molva et al. [MOLV85]. The case study is a fault tolerant virtual ring for bus allocation. With this protocol, a single entity is primary which means that it controls bus access. After a given period, the primary entity passes a privilege token to its logical successor. The claimed protocol service properties are:

- 1) Mutual exclusion - At any time, only one entity has the possibility of controlling bus access, ie. being primary.
- 2) Robustness - The primary status can be temporarily lost but will be recreated within finite time.
- 3) Fairness - Every non-faulty entity will in turn receive primary status.

Note that these properties are informal, they reflect service specifications. This is expected as MAC protocols typically lack a formal definition. These properties were monitored by a computer (observer) that was physically connected onto the bus as an additional entity through a normal hardware interface. Molva calls protocol entities *workers* and the testing hardware/software *observer* following the terminology of [AYAC79]. The global protocol state was encoded into a Petri net model for observer use. The observer monitored other stations by "spying" on workers as a passive observer. These observations indicated state changes and were compared with valid behavior from the Petri net model. This experiment was successful, a table in this paper lists observer detected errors. These errors were due to hardware problems, the observer located defective hardware. Molva describes the discovery of errors as follows:

These faults were not visible for a human observer since the service provided to the users of the communication system was perfectly correct. The observer was really useful because even if the external behavior of the experimental system under control seemed to be correct, the accumulation of such not visible faults could induce errors on the external behavior of the system, i.e. on the provided quality of service.

Conformance testing with observers in simulated environments follows along similar lines. Simulation (software) observers can also be either passive or active. The "test architecture" of observers in simulated systems is usually more powerful than that of figures 5.1, or 5.2. For example, observers in Véda [JARD88] or LANSF are equivalent to global observers with upper and lower testers at each entity. These observers have advantages not available with hardware observers that include:

- 1) Instantaneous knowledge of events - There is no transmission delay from testers to test harness. This simplifies protocol modelling within the harness.
- 2) Complete implementation access - Hardware IUTs are typically black boxes that can only be accessed via service access points. Simulation observers can validate internal consistency.
- 3) "Hardware" control - A simulation system models the communication medium and lower protocol layers. It can fake any error in these components to examine protocol response.

Véda [JARD88] is a FDT simulator based on Estelle. An earlier description of this system and its observer facility are found in [GROZ86]. For those interested, a recent technical note [VEDA89] (written in French) describes Véda observers in more detail. These papers consider a simple protocol involving two separate clocks. The service property is that these communicating clocks do not drift apart beyond a prespecified limit. A specification is produced in which the clocks test their consistency and re-synchronize when needed by sending messages. The specification claiming to satisfy the property is monitored and confirmed with an observer. This example illustrates issues related to software observation, for example what should be observed. With Estelle specifications, it is desirable to monitor process states, variables and interactions. To designate observed objects, Véda uses the concept of *probes*. Observer probes are defined at compile time to bind observed objects to observers. This is useful for the nested structure of Estelle specifications as it eliminates the need for long pathnames at run time and filters out additional objects with identical names from observation. In this manner, Véda observers can easily test invariants on monitored objects.

A second issue considered in these papers, primarily [GROZ86] is related to expressing service properties. The MAC protocol of Molva [MOLV85] and clock example both informally describe service properties. Formal service properties are preferable for the usual reasons; they eliminate interpretation ambiguities and permit mechanical translation into observers. The authors of Véda considered regular

expressions, temporal logic and other formal languages as candidates to express service properties. It was concluded that these languages were inadequate to express desired properties so the Estelle language was also chosen to specify protocol properties (ie. to program observers). With its underlying Pascal structure, Estelle permits the expression of any recursively enumerable property. Thus, these observers can check any property regarding the state of a Turing machine which includes all protocol implementations.

5.2. CSMA/CD Observers

To monitor CSMA/CD protocols, observers described in the previous section are not sufficient. As stated earlier, work related to FDTs cannot address MAC channel phenomena. In addition, MAC hardware observers [MOLV85] lack features to monitor contention based protocols. To illustrate, consider the case of a TCR collision. A hardware observer hears the collision but cannot discern whose packets are involved. Thus, tournament participants cannot be known in advance to such observers. It is here that the full power of simulation observers is useful. These observers can request notification when stations transmit and keep a log of current transmissions to register tournament participants when collisions occur.

Consider next the view that CSMA/CD observers should implement. An observer view is a partial or complete description of the global protocol state Φ . A view considered in chapter 1 is the product of local states $\phi_0 \dots \phi_{N-1}$. This view was judged unsatisfactory because of its state space explosion. It is also awkward to express channel contents with this view. The DP time line (figure 2.5) shows this view is unrelated to the human view of semi-controlled CSMA/CD. It is then presumably difficult to reason about protocol properties in this view. So what alternate view should CSMA/CD observers implement? Figure 2.5 provides an answer. In the human view, a semi-controlled CSMA/CD protocol has global states which include the following high level major states augmented by context information.

- 1) Uncontrolled silence
- 2) Uncontrolled transmission
- 3) Collision
- 4) Controlled silence (ie. unused slot)
- 5) Controlled transmission

These major states are given the notation Φ_i . The observer global state Φ is then expressed as one of the Φ_i with context information ρ . There is no claim that ρ contains all the network information, rather it encodes the minimal amount to test service properties. These high level states are an alternate description that correspond to a (potentially infinite) subset of global states expressed as local products. For example in TCR, uncontrolled silence includes the state with all Φ_i equal to IDLE. This alternate network state description using observers is a key point of this thesis. *Each observer describes the combined state of all network entities using a single major state Φ_i .* Observer high level states lead to synchronization questions. Given that stations perceive the same event at different times, when should the observer change states based on an event? For example, if a station starts an uncontrolled transmission should the observer switch states immediately or wait until all stations have heard the packet and agree to defer to this packet? These decisions are considered in the next chapter.

The last question is given an observer global view (Φ_i, ρ) , how do observers test implementation conformance? Presumably, the network global state has a temporal evolution given by $\Phi(T) = (\Phi_i, \rho^*)$ where ρ^* contains all of the context information. Observer verification corresponds to testing if $\rho^* \supseteq \rho$, it confirms that a subset of the context information is valid for the corresponding major state. To demonstrate by example, consider TCR service properties listed in section 4.3. These properties describe constraints on the observer context information ρ within specific major states Φ_i . For example, during a controlled slot unprivileged stations cannot transmit. Context information during the corresponding Φ_i should include information such as the privileged subtree T^* and station subset S_t attempting to seize the slot. An observer testing this property asserts the relation on ρ (ie. $T^* \supseteq S_t$) at some point during this major state. With LANSF, context information assertions occur immediately following each observer state transition. Before considering the form of these assertions, a description of how LANSF observers perform state transitions is presented in the next section. Conformance tests follow in the next chapter.

5.3. LANSF Observers - Model and Specification

LANSF observers are tools that validate global properties of LANSF protocol implementations. Their specification language is C, which means they can express any recursively enumerable property. As with LANSF protocols, there are no mechanical methods to generate observers. Each observer is user-programmed to assert an informal service property. The environment of LANSF observer processes includes station and generic LANSF processes. They are passive observers designed to dynamically monitor but not interfere with protocol simulations. The exception occurs when observers detect an error, in which case they terminate the simulation and print a short diagnostic. LANSF observers have unlimited access to simulation data structures. They assert the consistency of station process states/variables, virtual time and their private data structures denoted by ρ . In principle, observers can modify simulation data structures but are not used here for that purpose. Further, the observers described here only examine data structures within the current station, that is, the station returning from a system wakeup call.

Observers do not respond to simulated network events and do not generate such events. Instead, they respond to *metaevents* which include station process transitions and the passing of virtual time. As LANSF transitions are atomic, observers cannot monitor stations while they change states. Observers use LANSF environment services to learn of metaevents, this environment is an indirect method of monitoring stations. The first metaevent type monitors the control state of station processes. These metaevents occur when a station process has set its transition conditions and returned control to the environment. The environment wakes up any observer requesting to monitor the station in its current state. Recall that no virtual time passes while a station performs its actions, so observers learn of stations entering a control state instantly. For the second metaevent type, observers request to be awakened after a specified delay.

LANSF observers are also modelled as communicating EFSMs. These processes are awakened by LANSF when an awaited metaevent occurs. Observers perform actions relevant to wakeup metastates (updating and asserting relations on ρ), then set new metaevent wakeup conditions and return control to LANSF. For efficiency, LANSF puts observer processes to sleep while they await metaevents. Observer actions validate station process correctness to the same extent that assertions on ρ insure service properties.

For example, in two party protocols an observer can test if both parties are waiting for the other to act.

When such an assertion is performed, observers detect this particular case of protocol deadlock.

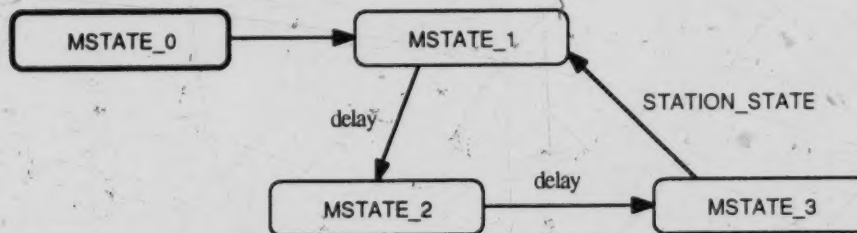


Figure 5.3 - States/transitions of a LANSF observer

LANSF observer control structure is similar to that of station processes, a sample state/transition diagram is shown in figure 5.3. Observers have one initial metastate MSTATE_0 indicated by a bold outline.

Transitions occur by three different mechanisms. Direct transitions have no prerequisites and are depicted by unlabeled arcs, the transition from MSTATE_0 is an example. These transitions precede pending station transitions and leave the observer environment unchanged. A second mechanism is the passing of virtual time. Transitions of this kind label arcs with plain text such as "delay" in this figure. The order of station and observer transitions set for the same virtual time is non-deterministic. The third transition type occurs when station processes enter observed control states. The transition from MSTATE_3 to MSTATE_1 is an example and corresponding arcs are labelled with the observed state (STATION_STATE). These transitions occur at the virtual time of stations entering the observed state. In this case, station transitions precede observer transitions and stations perform state actions before control passes to observers.

Observers are typically created during the initialization phase along with station processes. To create an observer, users call `new_observer (code)` where `code` is a pointer to the observer function. Figure 5.3 suggests that the LANSF observer structure is similar to that of station processes, this is indeed the case. The code of observer processes is a single function with a switch statement based on global variable `the_observer_action`. The generic code structure of an observer process is given below.

```

observer_process() {
    switch (the_observer_action) {
        case INITIALIZE:
            init_actions;
            init_transitions;
            return;
        case METASTATE_1:
            metastate_1_actions;
            metastate_1_transitions;
            return;
        ...
        case METASTATE_N:
            metastate_n_actions;
            metastate_n_transitions;
            return;
    }
}

```

Metastate actions have been discussed previously, they are operations on context information p . Metastate transitions are set on the basis of two metaevent types and direct transitions. The first metaevent class to monitor station states is supported by LANSF function `inspect (s, p, v, a, ma)`. The first parameter indicates which station(s) to monitor. Our observers that are summarized in Appendix D monitor all stations as indicated by constant ANY. The second parameter names the observed station process, only `transmitter` is monitored here. Next comes the process version number which is not used in our examples and is set to ANY. The fourth parameter names the observed station control state. There are several monitored states in the TCR implementation. A final parameter indicates the state to which observers branch when the metaevent occurs. The second metaevent type is the passing of virtual time and uses function `timeout (delay, ma)`. It has the wake-up delay and new observer state as parameters. Finally, observers perform direct transitions with LANSF function `resume_at (new_state)` to simplify their code.

A few general comments regarding the implementation of LANSF observers are in order here. These observers do not drive the simulation, hence they provide no guarantee of exploring any particular state. In LANSF, network events are initiated by the CLIENT A_i which produces packets for delivery. This A_i is programmable and can be modified to generate specific event sequences. However, there is no standard communication between this A_i and observers. To learn of network events (the behavior of CLIENT), observers are dependent on station processes entering states which indicate such events. This dependency assumes a certain degree of implementation liveness for observers to operate. Without this liveness, observers have no knowledge of network events and cannot modify their major state and context information

accordingly. This is not a serious limitation on protocol development: observers should only be utilized to verify an implementation that has previously demonstrated liveness. The lack of standard communication from observers to CLIENT is a more serious limitation. It precludes dynamic state exploration that is driven by observers on the basis of the current state or other information. Such exploration could be used to produce stressful situations for the protocol in which the probability of errors is increased.

Chapter 6

Verifying CSMA/CD-TCR

This chapter presents LANSF observers that verified the initial CSMA/CD-TCR implementation and the testing results. In chapter 4 the following controlled mode service properties were claimed for this protocol:

1. Stations transmit only when they are within the privileged subtree.
2. Any station not participating in an uncontrolled (controlled) collision will not transmit during the subsequent tournament (sub-tournament).
3. All stations participating in an uncontrolled (controlled) collision successfully transmit during the subsequent tournament (sub-tournament).

Conformance testing was used to search for implementation non-conformance to these properties. A separate observer was programmed for each property, their names were `check_privilege`, `check_participant` and `check_tournament` respectively. Testing revealed incorrect behavior of protocol entities relative to an observer model while CLIENT drove the network through a random state space exploration.

The claimed properties relate to the question: *Which stations have the privilege or responsibility of accessing the channel?* For testing purposes, observers maintain a global state Φ_i and context information ρ to indicate their view. The primary context information is the current privileged subtree T^* which is applicable to all 3 properties. The first property is directly tested, a check examines if a transmitting station is a member of T^* . The second and third properties use T^* but require additional observer context information when tests are performed. For example, the start and finish of tournaments are determined by changes to the privileged subtree.

The first section of this chapter describes the common observer state model, that is, the major states Φ_i and how the privileged subtree is maintained. Section 2 examines the LANSF code for the first observer and explains changes needed to obtain the other observers. The complete observer code is listed in Appendix D. The final section summarizes results of experiments performed with these observers.

6.1. CSMA/CD Observer States/Transitions

The observer view (Φ_i, ρ) corresponds to a partial state description that is orthogonal to that of stations. The tests performed with this view use the idea that a station subset (T^*) always has channel privilege and stations not within T^* are forbidden channel access. The initial observer goal is to correctly maintain the pair (Φ_i, ρ) . Additional code allows observers to test protocol properties. This section considers how observers maintain the major state Φ_i and context information $\rho = T^*$ that is applicable to any semi-controlled CSMA/CD protocol. Consider first the observer model of uncontrolled CSMA/CD mode. There are no claimed service properties for this mode but observers track its events to correctly switch to controlled mode. Our observer model treats uncontrolled mode as a special case of controlled mode to simplify the control structure. Uncontrolled mode is viewed as a series of virtual slots in which the entire tree is privileged. Figure 6.1 depicts a time line with these alternate views of uncontrolled mode. Note that protocol entities do not implement uncontrolled mode slots. If that were the case they would transmit only at the start of slots during this mode. Such behavior violates CSMA/CD uncontrolled mode specifications. Virtual slots are observer constructs used to present a systematic view of both modes.

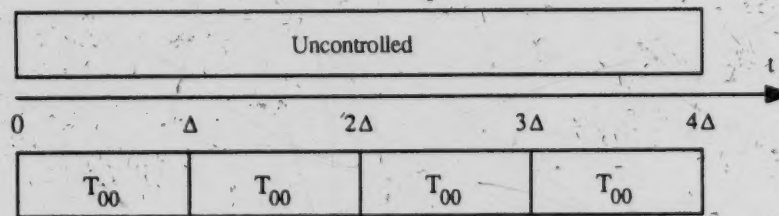


Figure 6.1 - Two views of CSMA/CD uncontrolled mode

Figure 6.2 shows the observer states used to track network slots, uncontrolled mode is a special case of this figure. Observers start in control state INITIALIZE and set the entire tree as privileged. From there, they perform a direct transition to SLOT_STARTING. Even at network startup (uncontrolled mode) observers initiate virtual slots. From SLOT_STARTING there are 2 transitions. The transition to SLOT_FINISHED occurs if there is no activity for a period slot_length which equals Δ presented earlier. For both modes SLOT_FINISHED terminates the current virtual slot. In uncontrolled mode, the terminated slot is replaced

with an identical slot as shown in figure 6.1. In controlled mode, the new slot has a different privileged subtree taken from the first reservation schedule item. After modifying the privileged subtree, state SLOT_FINISHED returns control directly to SLOT_STARTING.

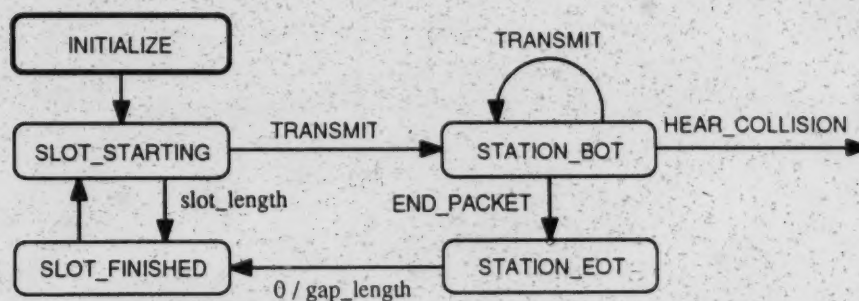


Figure 6.2 - CSMA/CD observer slot states

The second observer transition from SLOT_STARTING transfers control to STATION_BOT. This occurs when a station enters state TRANSMIT or equivalently starts transmitting a packet. A consequence of this transition is that the slot cannot time out as there are no timer transitions from STATION_BOT. This precludes observers testing local constraints such as a maximum packet length. Such tests however are trivial, the objective here is to validate global synchronization constraints. Observers leave STATION_BOT when the packet is successfully transmitted or a collision is reported (state HEAR_COLLISION). Following a successful transmission, observers proceed to STATION_EOT. From this state, control transfers to SLOT_FINISHED immediately for uncontrolled mode. In controlled mode, there is an inter-packet gap preceding the transition. An immediate transition is needed in uncontrolled mode to insure that observers reach SLOT_STARTING before any station starts another packet. Otherwise, an uncontrolled transmission could commence unknown to observers. In controlled mode, a short delay at the start of slots precludes this possibility. State SLOT_FINISHED again terminates the current slot and transfers to SLOT_STARTING. If a collision occurs with the observer in state STATION_BOT, control switches to COLLISION_HEARD which is discussed later. A third event, the start of additional packets is also awaited by state STATION_BOT. This produces a transition back into the same state and guarantees a collision soon afterwards. This additional transition insures observation of all attempted packet transmissions.

The previous observer state loops, for idle slots and successful transmissions, leave the network in uncontrolled mode. In this mode the time at which observers enter `SLOT_STARTING` is irrelevant. As shown in figure 6.1, uncontrolled virtual slots are an alternate view of unrestricted channel access. When uncontrolled transmissions commence, observers simultaneously transfer to `STATION_BOT`. For successful transmissions, observers enter state `STATION_EOT` simultaneously with the packet termination. As will be seen, synchronizing observers with the first station to know of events (transmissions, collisions, etc.) simplifies observer control structure. However, this requires instantaneous communication and is only possible with software observers. In controlled mode, there is always synchronous behavior between observers and stations. For this mode, observers enter `SLOT_STARTING` *simultaneously with the first network station that commences the slot*. This occurs after clearing of collisions, timeout of controlled slots or termination of successfully controlled transmissions. To maintain perfect synchronization through multiple slots, stations require errorless local clocks. This situation does not exist, the next chapter examines the corresponding observer-station synchronization relation with imperfect station clocks.

The balance of observer states monitor networks through the resynchronization period following collisions. Figure 6.3 shows the states/transitions involved in this process.

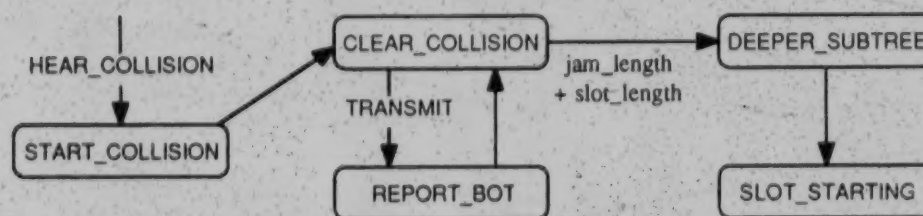


Figure 6.3 - CSMA/CD observer collision states

Observers enter `START_COLLISION` when a station (called the leading station) first hears a collision. The time at which this station starts the first controlled slot (ie. collision cleared) is logged by this observer state. Observers enter `SLOT_STARTING` at this same time (ignoring clock errors) to remain synchronized with the leading station. In the interim, observers transfer control to `CLEAR_COLLISION`. This state along with `REPORT_BOT` observes additional transmissions after the collision occurs but before it reaches

all stations. Following the clearing delay control transfers to DEEPER_SUBTREE which adjusts the privileged subtree for TCR. In other semi-controlled protocols, this state would adjust the privileged station subset according to its slot allocation algorithm. Once the observer has adjusted the privileged subtree, control passes directly to SLOT_STARTING.

If observers are synchronized with the leading station during the first controlled slot and both view slots with equal periods (ignoring clock errors), it follows that they remain synchronized through successive idle slots. However, when a station seizes a slot, the transmitting station is the first to know that it commences and finishes the packet. For this situation observers resynchronize with the transmitting station which becomes the new leading station. Resynchronization occurs when observers enter states STATION_BOT and STATION_EOT as transmitting stations enter TRANSMIT and END_PACKET respectively, this is shown in figure 6.2. After terminating a packet, transmitting stations wait through a delay gap_length before starting the next controlled slot. For controlled mode, observers have a similar delay in STATION_EOT preceding their return to SLOT_STARTING. To summarize, observer states have three main functions; timing idle slots, tracking transmissions and resynchronizing after collisions. The major states Φ_i start and terminate these functions as well as updating p . The context information p stores associated information which always includes the privileged subtree.

6.2. A Sample Observer

This section examines LANSF observer check_privilege in detail. It tested the first CSMA/CD-TCR property; only stations within the privileged subtree access the channel. Figure 6.1 shows that observers treat uncontrolled mode as a special case of controlled mode. This view is enforced by check_privilege, it examines all transmissions to insure that transmitting stations are privileged. However, it is impossible to generate exceptions in uncontrolled mode as all stations are privileged.

The only context information maintained by this observer is the current privileged subtree ($p = T^*$). In our implementation, T^* is represented by an integer tree_depth and integer array tree. The elements of tree are binary values: 0 (LEFT) or 1 (RIGHT). Variable tree_depth is initialized to 0 which indicates the

entire tree is privileged. During controlled mode, the array elements $0 \dots \text{tree\_depth} - 1$ form a binary representation of T^* . For example, if tree_depth equals 2 and $\text{tree}[0]$ ([1]) is LEFT (RIGHT), T^* is at level 2 and located by a left then a right branch from the root which is equivalent to T_{21} (figure 4.2). Stations are placed in subtrees according to their binary LANSF identifier which lies between 0 and $N - 1$. The method used differs from that indicated in figure 4.2; the low order bit of station identifiers is given highest priority. That is, stations with an even identifier (last bit 0) are within subtree T_{10} and those with an odd identifier (last bit 1) are members of T_{11} . This reversal of bit priorities simplifies the problem of accommodating variable length identifiers.

Chapter 4 suggests that a reservation schedule of privileged subtrees is maintained during controlled mode. This is not the case, the privileged subtree for the next slot is calculated from the current T^* . Two operations change the privileged subtree. The first occurs after collisions and is implemented by function `deeper_subtree` in state `DEEPER_SUBTREE`. For this case the privileged subtree becomes the left subtree of the previous T^* , ie, $T^* = L(T^*)$ where $L = \text{left child}$. The second operation is performed when slots successfully terminate after a timeout or end of packet. The corresponding change to T^* is performed by function `next_subtree` within state `SLOT_FINISHED`. Conceptually this function pops the reservation stack and sets the privileged subtree from the top item. Equivalently the *successor* node from the tournament tree becomes privileged, $T^* = S(T^*)$. The successor is inferred in the following manner. The simplest case is $T^* = T_{00}$ as indicated by tree_depth equal to 0. In this situation the stack is empty, ie, the protocol is in uncontrolled mode and T^* does not change. A second case occurs when tree_depth exceeds 0 and the least significant element of tree is LEFT. Recall that slots are allocated in pairs, the first to the left and the second for the right child of the previous subtree. The only slots that can interleave between siblings are descendants of the left sibling. However, such slots require a collision in the left sibling slot and cannot appear after a successful transmission. Hence if T^* is the left child of a node, its depth is unchanged during the next slot, only its last element changes from LEFT to RIGHT. The last case involves tree_depth not equal to 0 and the last element of tree equal to RIGHT (ie, $T^* = T_{x+1,2y+1}$). In this situation, two siblings ($T_{x+1,2y}$ and $T_{x+1,2y+1}$) have been made privileged to resolve a collision in their parent T_{xy} . The successor to $T_{x+1,2y+1}$ is then the successor of its parent. If T_{xy} is one of the cases

described previously; its successor is known. Otherwise T_{xy} also has $tree\_depth > 0$ and its last element is RIGHT so the successor process is applied recursively.

Observers update T^* in states DEEPER_SUBTREE and SLOT_FINISHED as discussed previously and shown in figures 6.2 and 6.3. Observer variations appear in tests performed with this information. Observer `check_privilege` tests the privilege of transmitting stations. The start of packets produce observer transitions to STATION_BOT (REPORT_BOT) when the observer has not (has) been informed of a collision. Both states call function `privileged_station` to assert the privilege of the transmitting station. For illegal transmissions, this observer terminates the simulation and reports the offending station.

The second and third TCR properties listed in the chapter introduction require an array P to maintain tournament (and sub-tournament) participants. For the corresponding observers, $p = T^* + P$. This two dimensional array is called participants. The first array index is the level; level 0 corresponds to tournaments (uncontrolled participants) and higher levels to sub-tournaments. There is at most 1 active (sub-) tournament for any level and the number of active sub-tournaments equals the depth of T^* . The level of network operation is always given by $tree\_depth$, observers register transmitting stations at this level. For example, uncontrolled transmissions are logged at level 0. The second index identifies stations; hence participants[depth][station_id] is TRUE if station station_id is currently a participant at level depth. Both observers use states STATION_BOT and REPORT_BOT to log participants. Instead of testing station privilege in these states, a call to `log_participant` records that station as a participant at level $tree\_depth$. State STATION_EOT calls function `clear_participant` to remove successfully transmitting stations as participants in all active (sub-) tournaments, from level $tree\_depth$ down to 0. Before performing this action, `clear_participant` ensures that the station in question was indeed a participant.

Observer `check_participant` tests the validity of stations to transmit in a sub-tournament based on properly registering at the previous level. For example, if $tree\_depth$ equals 1, the privileged subtree is either T_{10} or T_{11} . Stations transmitting with either value of T^* must have participated in the level 0 collision, that is, when T_{00} was privileged. This test is performed in STATION_BOT and REPORT_BOT before stations are logged for level $tree\_depth$. Note that stations participating in a level M collision are those that started transmitting at that level and did not successfully complete the transmission, hence their

logged-on status is still active. Obviously, there is no prior registration required to transmit in uncontrolled mode. However, for sub-tournaments, stations must have registered in the previous level to be valid participants at the current level.

Observer.check_tournament uses the participant array to test the third TCR property, that all stations who log themselves successfully transmit during a (sub-) tournament. The most natural place to test this property is within function next_subtree which is used to terminate slots. Note that a (sub-) tournament is finished when this function decrements the value of tree_depth. Hence, a test can be placed in this section of code for remaining participants, this is performed by function level_empty. The modification to next_subtree for this function is as follows:

```

if (tree [tree_depth - 1] == LEFT)
    tree [tree_depth - 1] = RIGHT;
else {
    tree_depth--;
    if (!level_empty (tree_depth)) excptn ('Incomplete tournament');
    next_subtree ();
}

```

To illustrate, consider an example in which stations 2 and 7 collide in uncontrolled mode. Both are registered as participants in level 0 prior to the uncontrolled collision. Station 2 seizes the first controlled slot and is registered at level 1 when it commences transmission. If another collision occurred at this point, station 2 would be legally registered to transmit with tree_depth equal to 2. However, the packet is successful so this station is deregistered at levels 1 and 0. After the second controlled slot, station 7 is also deregistered from both levels. The testing procedure does not check for "completed tournaments" in the leaves of tournament trees, only interior nodes are tested. It is evident that a slot without a collision cannot leave stations that commence but fail to terminate transmissions. As a consequence, level 0 following successful uncontrolled transmissions is also not tested.

6.3. Results of Observer Conformance Testing

Let us now enumerate errors (or inconsistencies) located by observers while conformance testing the initial implementation. These are not listed in their discovery order, instead they are ordered for pedagogical clarity. The simplest error was reported by observer `check_participant`. In its initial version, this observer tested whether stations transmitting during controlled mode had participated in the previous uncontrolled collision. According to `check_participant`, this condition was not met. This reported error was not due to an incorrect implementation, rather it was an interface error between the implementation and observer. The problem was that station processes performed direct state transitions which were invisible to the LANSF environment and consequently observers. This is possible because of C switch statement semantics. In C it is syntactically permissible to omit a `return` or `break` statement at the end of case constants. As seen in Appendix C, the code of state IDLE has the following form.

```
case IDLE:
```

```
    if (idle_period = current_time - last_silence < gap_length) {  
        wait_event (DELAY, gap_length - idle_period, TRANSMIT);  
        return;  
    }
```

```
case TRANSMIT:
```

When control reaches the final `if` statement of IDLE and this test fails, control continues into TRANSMIT. This constitutes an invisible, albeit efficient transition from IDLE to TRANSMIT during uncontrolled mode. The LANSF environment and observers are ignorant of this transition. However, in controlled mode all transitions to TRANSMIT are visible to the LANSF environment, hence observers are aware of all controlled transmissions. This leads to the result that some controlled transmissions are considered illegal by this observer because it failed to see the previous uncontrolled attempt. The solution to this problem is simple, all transitions must be externally visible. This is accomplished by explicitly terminating control states with a `continue_at (new_state)` statement as required. This modification affects states INITIALIZE and IDLE in Appendix C. For this protocol, the direct transition from INITIALIZE is of no concern because entries into state IDLE are not monitored. However, it is a good software engineering practice when working with observers to explicitly perform all station transitions.

Once this change was made, check_participant reported problems with deregistering stations after they finished a transmission. To illustrate the source, consider the following example. Station *J* successfully transmits in uncontrolled mode. Observers are instantaneously aware of this fact and enter state STATION_EOT (figure 6.2). A nearby station *K* soon hears this end of packet and starts its own transmission after enforcing a gap. In the meantime, observer control has progressed to SLOT_STARTING and the packet from station *K* switches observer control to STATION_BOT. A third distant station *Z* does not hear the end of packet from *J* until after *K* has started its transmission. This is because transmission delays are simulated for network stations but not for observers. When *Z* hears the end of packet from *J*, it enters state END_PACKET. The observer assumes this to be the end of packet for station *K*, as it has the only current packet visible to the observer. However a test of the observed station identity contradicts this assumption. As expected this situation only occurred in large networks where the channel length exceeded the inter-packet gap. With small networks, all stations hear the end of a packet before any station can begin another packet. In large networks observers with their instantaneous knowledge can be misled by stations reporting events considerably later. The problem here is the multiple meaning of state END_PACKET. This state was entered by transmitting stations when a packet was completed and by listening stations when they heard the end of packet. A simple solution is to differentiate transmitting and listening stations with two different states, this is shown in the second implementation. As with the previous inconsistency this was not a semantic implementation error, it was an interface error between implementation and observers. It re-iterates a second software engineering principle: all states should have a single meaning.

After making changes to overcome interface errors, observer check_participant discovered a minor semantic error. The implementation permitted channel access during a tournament to stations not involved in the uncontrolled collision. However, *offending stations had their packets at the start of tournaments*. This error was due to the implementation always forbidding stations from transmitting in uncontrolled mode when they heard channel activity; see state IDLE in Appendix C. The sample error discovered was as follows: It occurred when stations 1 and 2 were in a tournament and stations 0, 1 and 2 were backlogged during that tournament. The first tournament ended with the transmission by station 2 which was then the first station to transmit in uncontrolled mode. This packet reached station 1 before the latter started its

transmission, a legitimate result of local clock errors. Station 1 thus refrained from transmitting. However, the clock of station 0 was fast, so it started to transmit before hearing the packet from station 2. A collision occurred and started the second tournament. Stations 2 and 0 registered for the tournament by producing the collision. Station 1 did not participate in the collision but had its packet buffered when the collision occurred. It then participated in the tournament and produced the observed error.

In the initial implementation, this error can only occur after tournaments. There is no problem with uncontrolled mode as stations enter IDLE immediately after hearing the end of packet. The channel is idle so backlogged stations buffer a packet and then set a timer to commence transmission after a gap without retesting channel status. However, after a tournament stations enforce the inter-packet gap before returning to state IDLE so local clock errors permit some stations to reach this state faster than others. This error has the advantage of permitting a higher throughput in uncontrolled mode. It also satisfies the constraint that stations with buffered packets at the start of tournaments successfully transmit during the tournament, which is closely related to property 2. The problem is not properly registering stations for tournaments; a constraint attempting to impose network fairness. This error is traced to state IDLE trying to perform too many operations and reinforces the principle that states should have one well defined meaning.

The last error, reported by `check_privilege` was packet transmission by unprivileged stations. This obviously occurs only in controlled mode, a sample error is shown in figure 6.4. At the start of this example, stations *J* and *K* have `delay_count` equal to 1 and 2 respectively and no other stations compete for slots. `Defer_count` equals 3 so stations revert to uncontrolled mode after station *K* transmits.

Observer	Unused	Station J	Station K	U	
IUT	Unused	Station J	Unused	Station K	U

Figure 6.4 - Observer vs. IUT view of slots

Assume this situation was reached following a successful controlled transmission of a third station *M*. In state `END_PACKET` of the original implementation, station *J* (*K*) sets a timer delay equal to a packet gap plus one (two) slots upon hearing the end of packet *M*. The intention of both stations is to seize the

channel after their delay has elapsed. The timer of station *J* finishes first so this station commences and successfully terminates its packet. The timer of *K* is stopped when it hears channel activity and not restarted until hearing the end of packet *J*. At this point, station *K* should simply enforce a gap and start transmitting. A code examination shows that station *K* decrements `delay_count` by only one unit! The result is that *K* resets its timer to wait through a gap and then an empty slot. For this example the protocol requires an extra slot to finish the tournament. However, stations *J* and *K* remain synchronized because neither station decremented their value of `defer_count` while waiting through the initial unused slot. Stations *J* and *K* (and other network stations) have the same perception of when the network returns to uncontrolled mode. However, their collective view is not consistent with that of observers.

Based on errors discovered in the initial IUT, the second TCR implementation and other semi-controlled CSMA/CD protocols should satisfy the following engineering principles and constraints:

1. All state transitions must be explicit.
2. States should have a single well-defined purpose.
3. The passing of idle slots must be noted by all stations.

The next chapter proposes a model for such an implementation. This model resembles the observer view and implementation constraints are expressed in terms of the observer metastates.

Chapter 7

Modelling Semi-Controlled CSMA/CD Protocols

The previous chapter presented an observer model to test the conformance of an initial CSMA/CD-TCR implementation. As was shown, observers turned up two interface and two semantic errors. These tests suggested general implementation guidelines and specific changes to the initial implementation. Another outcome of observer testing results from the fact that their specification required a global model. The correctness of a global model is not dependent on external variables in contrast to implementation models. For the latter, correctness proofs typically rely on assumptions of correct behavior at other entities. This introduces circular correctness proofs; entities must a priori behave in a certain fashion to prove that their behavior is correct. However, with observers correctness is not dependent on external variables; observer models define global states. By referring to a "correct" global observer model it is possible to prove implementation properties within this framework.

This chapter presents an abstract model for a second implementation version of the TCR protocol. It incorporates the lessons learned from observer testing and is applicable to all semi-controlled CSMA/CD protocols. To be specific, it shows the behavior of station process transmitter. The next chapter presents a LANSF implementation of this process. In this chapter, the observer-station state relations are examined in detail to prove that stations behave correctly relative to the observer model.

The first section of this chapter models the uncontrolled mode of CSMA/CD and shows the relationship of station states to observer states within this mode. The next section examines constraints associated with local clock errors for synchronized retransmissions. It then proposes a new model for controlled slots to enforce these constraints. Section 3 models the retransmission mode of semi-controlled protocols and again relates station states to observers. The last section provides a partial correctness proof for the slot model presented in section 2.

7.1. The Uncontrolled Mode of CSMA/CD Protocols

Uncontrolled mode contains behavior common to all CSMA/CD protocols. Its essential feature is that backlogged stations transmit as soon as the bus becomes idle and an inter-packet gap is enforced. A specification of this mode can be extracted from figure 3.3. This figure shows states used for an Ethernet transmitter; uncontrolled mode is obtained by ignoring states HEAR_COLLISION and STOP_JAM. This specification is sufficient for uncontrolled CSMA/CD protocols. However, the TCR specification pointed to a need for a more complex uncontrolled mode (figure 4.5). With semi-controlled protocols, all stations monitor uncontrolled transmissions for the purpose of switching to controlled mode in case a collision occurs. Figure 7.1 depicts uncontrolled mode for semi-controlled CSMA/CD protocols. A minor modification makes it acceptable to uncontrolled protocols. This change removes the collision transition from CHANNEL_BUSY and replaces the *end of packet* transition with a *bus silent* event. Given these changes, listening stations remain idle while the channel is active and resume competing for the channel after it has become silent; this defines an uncontrolled protocol:

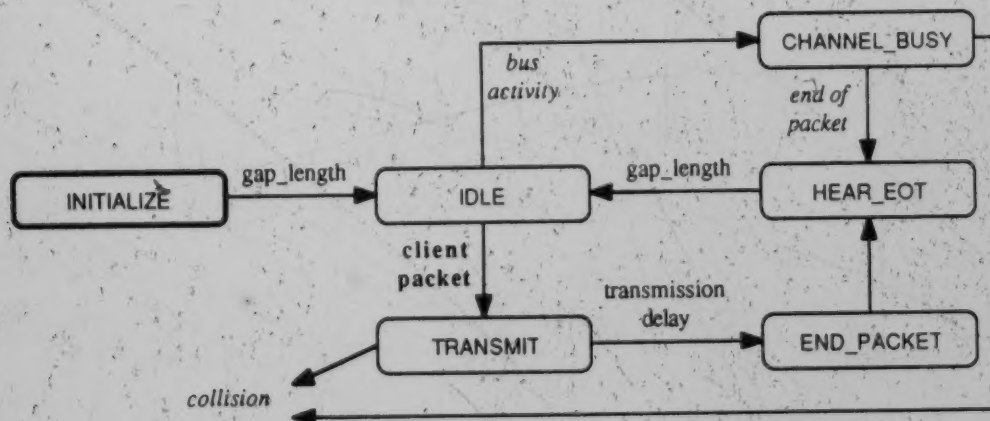


Figure 7.1 - CSMA/CD uncontrolled mode

Station transmitters commence their operation in state INITIALIZE and transfer to IDLE after an initial inter-packet gap. Upon entering IDLE stations first test if they are backlogged. This order insures that stations with a packet upon entering IDLE attempt to seize the channel. When no packet is available immediately,

stations wait for channel activity or client packet arrival. If a packet arrives, stations do not re-test the channel status before proceeding to TRANSMIT. This insures that stations cannot enter CHANNEL_BUSY in uncontrolled mode with a buffered packet ready for transmission. Hence, stations do not enter into tournaments with a packet yet having failed to participate in the initial collision.

With a buffered packet, stations branch to TRANSMIT and immediately start transmitting the packet. A successful transmission leads to END_PACKET which terminates the transfer and passes control to HEAR_EOT. HEAR_EOT is used to force an inter-packet gap before returning to IDLE. The transition to IDLE represents the end of one slot and the beginning of the next slot. For the transmitting station, its transition to HEAR_EOT occurs immediately from END_PACKET. Stated equivalently, the transmitting station hears its end of packet instantaneously. When non-transmitting stations hear channel activity they transfer control to CHANNEL_BUSY. This state waits for a successful packet termination or a collision. In the first case control switches to HEAR_EOT before returning to IDLE. CHANNEL_BUSY considers the possibility of collisions which insures that stations switch to controlled mode when such an event occurs.

Consider now a time line for observers and stations during a successful uncontrolled transmission. As stated earlier, observers synchronize with transmitting stations during an uncontrolled "slot" seized by a single station. If a transmission starts at observer time T , the transmitting station and observer have simultaneous transitions to states TRANSMIT and STATION_BOT respectively. Listening stations switch to state CHANNEL_BUSY within a period $L + S$ equal to channel length plus physical signalling delay. Including clock errors the transmitting station proceeds to END_PACKET and HEAR_EOT at $T + P'$, where P' is the packet length in observer time. Observers simultaneously switch to STATION_EOT and immediately proceed through SLOT_FINISHED and SLOT_STARTING to commence a new slot. The transmitting station returns to IDLE at $T + P' + G'$ where G' is the inter-packet gap including local clock errors. Listening stations follow through states HEAR_EOT and IDLE shortly afterwards based on the physical signalling delay and their distance from the transmitting station.

As stated earlier, it is not the goal of this thesis to verify uncontrolled mode. The model of figure 7.1 is considered correct on the basis of previous research and the only "correctness" criteria at this point is its interface with LANSF observers. The only problem that had to be resolved for uncontrolled mode was the

possibility of invisible transmissions, that is, stations commencing packets unknown to observers. If observers wait in STATION_EOT for a period equal to G , they can miss the start of new packets for $G' < G$ in the transmitting station. It would also be possible for nearby stations with fast clocks to start premature transmissions. For this reason, there is no observer delay in STATION_EOT, observers return immediately to SLOT_STARTING. The fact that observers break synchronization with transmitting stations at this point is of no concern for uncontrolled mode. The objective of this mode is for observers to enter controlled mode synchronized with transmitting stations and to insure that all transmissions are observed.

7.2. Slot Synchronization for Semi-Controlled Protocols

Retransmission slots were considered in the Ethernet and TCR implementations. It was shown that inexact slot timing was acceptable for uncontrolled protocols. However, for semi-controlled protocols, it is critical that stations consistently interpret retransmission slots. This section considers how to maintain synchronized slots in an environment with local clock errors and finite inter-station distances. This model is used for semi-controlled retransmission mode in the next section.

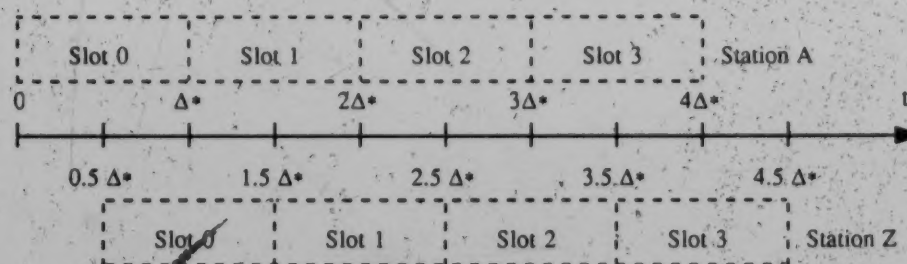


Figure 7.2 - Different station views of slots

Let us review some ideas of chapter 2. To simplify this discussion assume there is no channel status delay in the physical layer, ie. $S = 0$. Given a channel length of $0.5\Delta^*$ the synchronization period, minimum packet length and slot length are all Δ^* . The worst synchronization case happens when events such as collisions occur at one end of the channel. Two stations A and Z at opposite channel ends perceive these events at times differing by $0.5\Delta^*$. Figure 7.2 displays how these stations interpret slots after such

events given perfect clocks and assuming the event occurs at time 0 in front of A. If station Z uses slot m, it starts transmitting at $T = (m + 0.5)\Delta^*$. This packet is sensed by station A at $(m + 1)\Delta^*$, the very instant at which A starts slot m + 1.

Consider now problems arising from the limited accuracy of local clocks. Fractional clock error is quantified by ϵ where $dT = (1 + \epsilon) dT_{\text{clock}}$. Fast clocks have negative ϵ , slow clocks a positive value. The termination of idle slots locally corresponds to $dT_{\text{clock}} = \Delta^*$, so the true slot length is $(1 + \epsilon)\Delta^*$. Figure 7.2 suggests two possible errors, a packet from station Z arriving a slot late at A or a packet from A that is early at Z. Both errors result from a slow clock at Z or a fast clock at A. For example, if Z has $\epsilon = +0.1$, it starts slot m at $T_m = (0.5 + 1.1 m)\Delta^*$. A packet inserted into slot 1 leaves Z at $1.6\Delta^*$ and reaches A at $2.1\Delta^*$. With an accurate clock, A assumes this packet was placed in slot 2. Similarly, if A has $\epsilon = -0.1$, it commences slot m at $T_m = 0.9 m\Delta^*$. Then, if A uses slot 3 the packet starts at $2.7\Delta^*$ and reaches Z at $3.2\Delta^*$. Station Z erroneously assumes this packet to be in slot 2 given an accurate clock.

The first (second) error can be corrected by a trailing (leading) guard interval Γ . The interval length depends on the maximum clock error ϵ_{max} and number of consecutive idle slots M. Consider the first error where packets sent from Z arrive a slot late at A. Append a trailing guard interval of length $\Gamma = \lambda\Delta^*$ to each slot and define the slot period as $\Delta = (1 + \lambda)\Delta^*$. The worst synchronization case is stations A and Z having clock errors $-\epsilon_{\text{max}}$ and ϵ_{max} respectively. With this situation, the true slot lengths at A and Z are $(1 - \epsilon_{\text{max}})\Delta$ and $(1 + \epsilon_{\text{max}})\Delta$ respectively. For this case, the following constraints are satisfied:

- a. At station A, slot m starts and finishes at $(1 - \epsilon_{\text{max}})m\Delta$ and $(1 - \epsilon_{\text{max}})(m + 1)\Delta$ respectively.
- b. For station Z, slot m starts at $T_m = 0.5\Delta^* + (1 + \epsilon_{\text{max}})m\Delta$.
- c. A packet transmitted from Z in slot m reaches station A at time $T_m + 0.5\Delta^*$ or equivalently at $\Delta^* + (1 + \epsilon_{\text{max}})m\Delta$.

Consider a packet inserted into slot m by station Z. For a correct interpretation at A, this packet must reach A before slot m finishes. Constraints a. and c. require: $\Delta^* + (1 + \epsilon_{\text{max}})m\Delta < (1 - \epsilon_{\text{max}})(m + 1)\Delta$. To simplify, substitute the identity between Δ^* and Δ to produce $(2m + 1)\epsilon_{\text{max}} < \frac{\lambda}{1 + \lambda}$. A rearrangement yields $\lambda > \frac{\sigma}{1 - \sigma}$ where $\sigma = (2m + 1)\epsilon_{\text{max}}$. This equation shows that there is no solution for λ when $(2m + 1)\epsilon_{\text{max}} \geq 1$. This is a result of appending a guard interval to each slot; for a large number

of consecutive idle slots the additional time consumed by guards creates an unlimited need for additional guard length. The initial TCR implementation did not have this limitation, it appended a guard only to the non-idle slot following a silent period. However, with realistic networks this singularity is not a limitation. For $\epsilon_{\max} \ll 1$, the denominator σ term can be ignored to produce the result $\lambda > (2m + 1) \epsilon_{\max}$. To provide a margin of error for this simplification, λ can be set to $(2m + 2) \epsilon_{\max}$ for the largest value of m which is $M - 1$ so $\lambda = 2M\epsilon_{\max}$. The trailing guard length is then given by $\Gamma = 2M\epsilon_{\max}\Delta^*$. As discussed earlier the dynamic priority protocol has M equal to $N - 2$ ($N - 1$ if the clearing slot is included) and TCR has at most $\log_2 N - 1$ consecutive idle slots for N station networks.

The problem of packets from station A reaching Z early is solved by a leading guard interval on slots. Figure 7.3 shows a slot with both intervals of length $\Gamma = \lambda\Delta^*$. Leading intervals have the same length as trailing intervals since the problems they address are symmetric. The total slot length given by Δ is now $(1 + 2\lambda)\Delta^*$ or equivalently $\Delta^* + 2\Gamma$ with λ and Γ expressed above. The objective with such slots is for stations to begin transmission when they enter the gray zone depicted in figure 7.3. Other stations then hear the start of packets within the same slot, possibly during a guard interval.

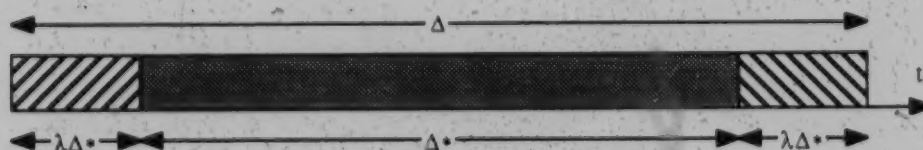


Figure 7.3 - Slot with guard intervals

A review of the initial TCR implementation shows that it incorporated the ideas depicted in figure 7.3. Chapter 2 stated that the maximum synchronization period Δ^* in the Ethernet standard was 450 bits. This standard also defines a minimum packet length Δ of 512 bits. Hence, two symmetric guard intervals would have lengths of approximately 31 bits each. These are sufficient to handle realistic clock errors. In the TCR implementation a short initial delay η is imposed by states REJOIN (DEFERRED) for stations wishing to access the channel during a controlled slot (or immediately after exiting controlled mode). This

is equivalent to the leading guard interval discussed above. The trailing guard interval is automatically provided by the extra length of slots. Its length implicitly equals $\Delta - \Delta^* - \eta$.

This slot model can be used to maintain eventless synchronization through M controlled slots. Afterwards the network must return to uncontrolled mode or produce a network event such as a collision or transmission. Another problem for synchronizing slots not mentioned above is that CSMA/CD packets are not of fixed length, only a minimum length Δ is imposed. This problem is not difficult to overcome as shown in the initial TCR implementation. When a station transmits or hears a packet, it dynamically extends the current slot until the packet is finished. Afterwards, a short delay is enforced to produce an inter-packet gap. This event resynchronizes network clocks and reduces the burden placed on the network by long periods of silence.

7.3. The Controlled Mode of CSMA/CD Protocols

In the semi-controlled CSMA/CD subclass, all stations switch to retransmission mode when collisions occur. This mode maintains slot synchronization so that consistent interpretations of privileged station subsets S^* are possible. Figure 7.4 shows the state diagram for controlled retransmission mode. Figure 7.1 indicates that transmitting (listening) stations enter this mode from state TRANSMIT (CHANNEL_BUSY) when collisions occur. All stations enter controlled mode into state HEAR_COLLISION. This state first allocates a run of slots and defines privileged station subsets for each slot. The implementation in figure 7.4 assumes that all stations jam the bus following collisions. This is a convention, the initial TCR had only transmitting stations jam the channel. State HEAR_COLLISION initiates jamming signals. After the jamming delay, control transfers to STOP_JAM. In this state, stations terminate jamming signals and delay through a channel clearing interval before proceeding to NEW_SLOT. This interval can be viewed as slot 0 (section 2.3). During this interval, observers are synchronized with the first station hearing the collision. Observers enter state START_COLLISION when the leading station enters HEAR_COLLISION.

The remaining states are described in context of the 3 slot cases: idle, used by a single station or used by multiple stations. The simplest case is an idle slot. Stations set a timer with delay Δ upon entering NEW_SLOT. If the channel is silent through this period, a transition occurs to END_SLOT at the end of the trailing guard interval depicted in figure 7.3. In END_SLOT, stations terminate the current slot and adjust their reservation schedule accordingly. This is the local equivalent of changes made by observers to S^* in SLOT_FINISHED. After slot termination there are two possibilities; all reserved slots have been executed or there remains additional slots. The first case produces a return to uncontrolled mode via state IDLE after a guard interval is enforced. The delay insures that uncontrolled packets are not misinterpreted/as packets inserted into the final controlled slot. The second case produces an immediate transition to NEW_SLOT to commence the next controlled slot.

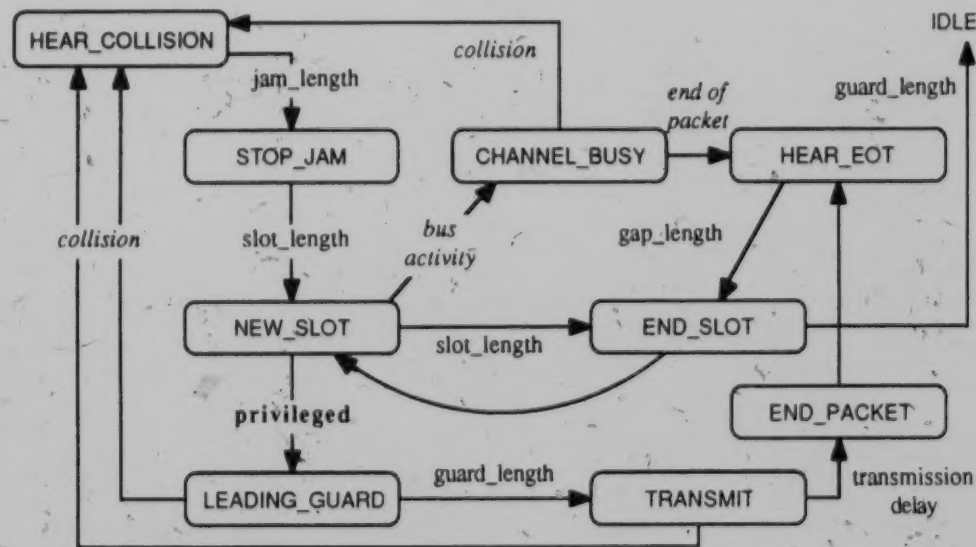


Figure 7.4 - CSMA/CD controlled mode

In the case of a single transmission during a slot, the transmitting station transfers from NEW_SLOT to LEADING_GUARD at the slot start. After a delay equal to Γ , this station branches to TRANSMIT and at the end of transmission to END_PACKET. From there control transfers to HEAR_EOT which enforces an inter-packet gap. Afterwards, state END_SLOT produces a transition to IDLE or NEW_SLOT. Non-transmitting stations hear the packet start in state NEW_SLOT and transfer control to CHANNEL_BUSY. This state has

two transitions, one for a successful transfer, the other for a collision. The first switches control to HEAR_EOT which enforces an inter-packet gap before transferring control to END_SLOT.

When multiple stations compete for a controlled slot, the collision is heard by every station. Transmitting stations are in state TRANSMIT when they hear the collision and listening stations in CHANNEL_BUSY. All stations proceed immediately to COLLISION_HEARD when they hear collisions. This state readjusts the reservation schedule and resynchronizes the network. The DP protocol produces no competition for controlled mode slots and such collisions never occur. In general, controlled mode collisions are possible and the reservation schedule is dynamically adjusted after such collisions. The Tree Collision Resolution (TCR) protocol illustrates an example of this behavior. The next chapter produces a LANSF implementation of the model illustrated in figures 7.1 and 7.4. This implementation is initially written for DP but can be easily modified to accommodate any CSMA/CD semi-controlled protocol. The needed changes for TCR are also given in chapter 8.

7.4. Timing Correctness of Controlled Mode Slots

Let us now demonstrate the consistency of slot interpretation as implemented by controlled mode states in section 3. To accomplish this goal, the timing constraints of stations will be proven consistent relative to the observer model. Consider a collision followed by $M - 1$ idle slots which produces M consecutive eventless slots including slot 0, the channel clearing slot. Define $T = 0$ when the leading station starts slot 0 and assume the observer commences this slot simultaneously. The problem of observer-station synchronization during the short jamming period prior to this slot is not considered. Finally let observer slots have length Δ so that $T_m = m\Delta$ is the time at which observers commence slot m . With these conditions, it can be shown that for $0 \leq m < M$ the following constraints hold:

- a. All stations enter slot m (state NEW_SLOT) no earlier than $T_m - 0.5 \Gamma$.
- b. All stations using slot m start transmitting no earlier than $T_m + 0.5 \Gamma$.
- c. All stations enter slot m no later than $T_m + 0.5 \Gamma + 0.5 \Delta$.
- d. Any transmission started in slot m is heard by all stations before $T_{m+1} + 0.5 \Gamma$.

Several consequences flow from these constraints. The first is that no station transmits in slot m before observers commence that slot, a result of the second constraint. Constraint b. is based on the first constraint, it follows trivially that no station can transmit in slot m before $T_m + 0.5 \Gamma$ as there is a delay of Γ from NEW_SLOT to TRANSMIT. A consequence of constraint d. is that transmissions are initiated and observed before the intended slot m terminates. Thus, the intended transmission slot is properly interpreted by observers. Of greater importance is whether intended slots are correctly interpreted by all stations. There are two incorrect possibilities, packets intended for slot m can be viewed within slots $m - 1$ or $m + 1$ by other stations. To prove that neither situation occurs, we will consider a simplified situation in which there is no signalling delay at the physical layer so that channel length equals $0.5 \Delta^*$. A non-zero signalling delay simply increases Δ^* relative to the channel length and this longer synchronization period compensates accordingly in the proofs. Further, assume clock errors are small so that the linear case in which $\epsilon_{\max} \ll 1$, $\Delta = (1 + 2\lambda) \Delta^*$ and $\lambda = (2M + 1) \epsilon_{\max}$ is applicable to proofs.

Consider the earliest and latest times at which stations commence slot m , as denoted by T'_m and T''_m respectively. These times for an arbitrary station are given by $m(1 - \epsilon_{\max})\Delta + \alpha$ and $m(1 + \epsilon_{\max})\Delta + \alpha$ where α is the propagation delay from the leading station. It follows that $T'_m = T_m - m\epsilon_{\max}\Delta + \alpha$ which is greater than $T_m - 0.5 \Gamma + \alpha$ since $\Gamma = 2M\epsilon_{\max}\Delta^*$. This directly proves timing constraint a. given above. Similarly, $T''_m = T_m + m\epsilon_{\max}\Delta + \alpha < T_m + 0.5 \Gamma + 0.5 \Delta^*$ which proves constraint c. The possibility of early transmissions is ruled out by constraints a. and c. No station can start transmitting before $T_m + 0.5 \Gamma$ so its packet does not reach any other station before $T_m + 0.5 \Gamma + \alpha$. All stations have commenced slot m by that time. Late packets also cannot occur because the latest a station starts transmitting in slot m is at $T_m + 1.5 \Gamma + \alpha$ including the delay to state TRANSMIT in slot m . This packet is heard before $T_m + 1.5 \Gamma + \alpha + 0.5 \Delta^*$ by all stations. This is less than $T_m + 1.5 \Gamma + \Delta^*$ which is equivalent to $T_{m+1} - 0.5 \Gamma$ and equals the earliest any station can terminate slot m .

This demonstrates the correct specification of controlled mode timing constraints and illustrates the usefulness of the observer model as a fixed reference frame in which to reason about implementation timing constraints. The proof above considered slots following a collision; a similar proof is easy to generate

following a successful controlled transmission. The final step is now to examine how this model can be specified with the LANSF implementation language, which is performed in the next chapter.

Chapter 8

LANSF CSMA/CD Implementations

In the previous chapter, a control state model was presented for semi-controlled CSMA/CD protocols. This chapter converts the protocol model into a LANSF implementation. The semi-controlled CSMA/CD implementation is written generically which leaves two undefined functions. Minor extensions then complete the implementation for any protocol in this class. The extensions for Dynamic Priority (DP) and Tree Collision Resolution (TCR) are presented here and Appendix E summarizes the TCR implementation. Simulations to test implementation conformance are performed for approximately 10^5 typical CSMA/CD packets under medium to heavy load. This packet limit occurs due to the implementation TIME data type but can be easily modified if longer simulations are desired.

The generic LANSF implementation of semi-controlled protocols has some common flags and variables to record the status of individual stations and the network. Network parameters are global variables and station information is added to STATION structures (Section 3.4). For timing controlled slots, stations require a global slot length Δ given by slot_length. This implementation uses the minimum synchronization period discussed in section 3.1 to test timing constraints under difficult conditions. Related quantities applicable to all protocols of this subclass include the guard interval Γ (guard_length), synchronization period (Δ^* or synch_period), maximum clock error ($\epsilon_{\max}$ or max_clock_error) and the potential number of consecutive idle slots max_idle_slots. In summary, the following definitions and initialization are added to the LANSF implementation of a generic semi-controlled CSMA/CD protocol.

```
int synch_period, guard_length, max_idle_slots, slot_length;
float max_clock_error;

struct STATION {
    ...
    int normal_mode, is_transmitting, is_competing;
    int defer_count, delay_count; };

```

```

initialization () {
    /* Read or calculate synch_period, max_idle_slots and max_clock_error */
    guard_length = 2 * max_idle_slots * max_clock_error * synch_period ;
    slot_length = synch_period + 2 * guard_length;
}

```

The additional local variables have the following purposes. The station protocol mode is kept in flag `normal_mode` and is initialized to TRUE. A flag `is_transmitting` initialized to FALSE indicates if the station is currently transmitting. It differentiates transmitting and listening stations as they enter the collision clearing phase. Flag `is_competing` indicates if the station has commenced but not yet successfully completed a transmission. This flag remains TRUE after a packet collides until a successful retransmission. It is a local equivalent of entries in the transmission matrix *P* maintained by observers (section 6.2). The semi-controlled model properties discussed in chapter 7 show that `is_competing` is equivalent to macro `transfer_pending` utilized in the initial implementation (Appendix C). This macro returned TRUE if the current station had a full buffer, flag `is_competing` is set and reset equivalently. The semantics of state IDLE initiate an uncontrolled transmission when a packet is buffered which sets `is_competing`. Similarly, this flag is reset and the packet released in state END_PACKET following a successful transmission.

The other two generic local variables are integers `defer_count` and `delay_count`. In controlled mode, stations maintain a local counter of remaining slots and when it reaches 0 stations revert to normal mode. This counter is `defer_count` and is not initialized by stations. During tournaments, stations waiting for a controlled slot require a slot counter to indicate when they are within the privileged subset. This slot count is stored in `delay_count` which is maintained when stations perform controlled mode and their `is_competing` flag is TRUE. These two variables contain local information that is collectively equivalent to the global reservation schedule.

Consider now the case study for converting the CFSM model into a LANSF implementation. In the first section, a generic implementation is given for the uncontrolled mode of semi-controlled protocols. The next section performs a similar task for their controlled mode. Section 3 fills out details of the generic implementation to complete Dynamic Priority. The fourth section solves this same problem for the TCR protocol which completes the goal of re-implementing TCR based on the observer model. In section 5, the experimental results obtained with the new TCR implementation and their significance are presented.

8.1. LANSF Normal Mode Implementation

Figure 7.1 shows the normal mode CFMS of semi-controlled CSMA/CD transmitters. Control state code can be informally derived from this model. State INITIALIZE initializes local variables and produces a delayed transition to IDLE. IDLE checks for a backlogged packet and if one is not available, it waits for a packet arrival or channel activity. This initial packet test is needed when multiple stations wish to transmit after the channel becomes idle. If station A detects the idle channel first, it waits through a gap and then starts transmitting. A second station B will hear the silence later and also wait through a gap. After this gap it may hear the packet from A (or a collision) at the same time or slightly before it intends to start transmitting. CSMA/CD states that station B must start its packet and force a collision in this situation, hence the initial test to give priority to backlogged packets. The code of states INITIALIZE and IDLE is shown below. In IDLE, there is no priority for simultaneous events within the else clause despite their lexical order. The key feature is that stations buffering packets are forced to state TRANSMIT so that stations in CHANNEL_BUSY during uncontrolled mode are guaranteed not to have packets. This guarantees that stations with packets at the start of tournaments participate in the uncontrolled collision.

```
case INITIALIZE:
    the_station->normal_mode = TRUE;
    the_station->is_transmitting = FALSE;
    the_station->is_competing = FALSE;
    wait_event (DELAY, gap_length, IDLE);
    return;

case IDLE:
    if (get_packet (BUFFER, min_length, max_length, info_length)) {
        continue_at (TRANSMIT);
    } else {
        wait_event (CLIENT, MESSAGE_ARRIVAL, IDLE);
        wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
        return;
    }
}
```

State TRANSMIT starts transmissions and waits for a result, either a successful transfer or a collision. It also sets flags is_transmitting and is_competing. Note that transitions from IDLE to TRANSMIT are simultaneous with packet acquisition. The code to implement TRANSMIT is the following:

```

case TRANSMIT:
    the_station->is_transmitting = TRUE;
    the_station->is_competing = TRUE;
    transmit_packet (BUS, BUFFER, END_PACKET);
    wait_event (BUS, COLLISION, HEAR_COLLISION);
    return;

```

For a successful packet, the transmitting station branches to END_PACKET and then to EOT_HEARD. The first state terminates the transmission and releases the packet. It also resets the station transmission status. When stations successfully terminate packets they no longer compete for the channel as indicated by resetting flag is_competing. The purpose of HEAR_EOT is to enforce inter-packet gaps and then transfer control to the appropriate state. In uncontrolled mode this state is IDLE, controlled mode is considered shortly. The passing of all stations through this state after successful transmissions proves the unobserved service property that requires inter-packet gaps.

```

case END_PACKET:
    the_station->is_transmitting = FALSE;
    the_station->is_competing = FALSE;
    stop_transfer (BUS);
    release_packet (BUFFER);
    continue_at (HEAR_EOT);

```

```

case HEAR_EOT:
    if (the_station->normal mode) {
        wait_event (DELAY, gap_length, IDLE);
        return;
    } else { /* Controlled mode */

```

```

case CHANNEL_BUSY:
    wait_event (BUS, EOT, HEAR_EOT);
    wait_event (BUS, COLLISION, HEAR_COLLISION);
    return;

```

Listening stations branch to CHANNEL_BUSY when they hear channel activity. For uncontrolled mode, stations in this state do not have buffered packets. Stations leave CHANNEL_BUSY when the packet is successfully terminated or when a collision is heard. Hearing the end of a successful transmission switches control to HEAR_EOT where these stations also wait through a gap. If a collision is heard, listening stations also switch to retransmission mode starting with state HEAR_COLLISION.

8.2. LANSF Controlled Mode Implementation

The CSMA/CD controlled mode introduces new control states and extends the operation of some existing states. The first controlled mode state is HEAR_COLLISION. This state aborts transmissions, starts a jamming signal and prepares for the upcoming (or continuing) controlled mode. Stations prepare for the tournament by adjusting their values of defer_count and for tournament participants delay_count.

```
case HEAR_COLLISION:
    if (the_station->is_transmitting) {
        abort_transfer (BUS);
        the_station->is_transmitting = FALSE;
    }
    set_defer_count ();
    if (the_station->is_competing) set_delay_count ();
    the_station->normal_mode = FALSE;
    emit_short_jam (BUS, jam_length, STOP_JAM);
    return;
```

After collisions variable is_competing remains unchanged, stations that have started but not finished a packet leave it set. The number of tournament slots is calculated by set_defer_count. For competing stations, the number of slots until they transmit is determined by set_delay_count. Stations then switch modes and jam the bus to enforce the collision. The transition from HEAR_COLLISION is to STOP_JAM. The latter stops the jamming signal and enforces a bus clearing delay before the first slot. Its code is:

```
case STOP_JAM:
    stop_jam (BUS);
    wait_event (DELAY, slot_length, NEW_SLOT);
    return;
```

After the idle slot enforced by STOP_JAM terminates, stations proceed to NEW_SLOT. Figure 7.4 shows 3 transitions from this state that correspond to an idle slot, a slot in which one station transmits or a slot in which multiple stations transmit. The code of this state is:

```
case NEW_SLOT:
    if (station_privileged ()) {
        continue_at (LEADING_GUARD);
    } else {
        wait_event (DELAY, slot_length, END_SLOT);
        wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
        return;
    }
```

To understand the transitions it is useful to examine the 3 slot cases. The simplest is an idle slot. In this situation, all stations fail the initial `station_privileged` test and set a timer equal to the slot duration given by `slot_length`. There is no activity during the slot so transitions to `END_SLOT` occur after the specified delay. State `END_SLOT` adjusts tournament variables to reflect an elapsed slot. If there are no remaining slots, stations revert to uncontrolled mode. If controlled mode is not finished, stations branch to `NEW_SLOT` and backlogged stations wait-through one less slot.

```
case END_SLOT:
    the_station->defer_count--;
    if (the_station->defer_count == 0) {
        the_station->normal_mode = TRUE;
        wait_event (DELAY, guard_length, IDLE);
        return;
    } else {
        if (the_station->is_competing) the_station->delay_count--;
        continue_at (NEW_SLOT);
    }
}
```

The second case for `NEW_SLOT` is a single transmission. The station at which `station_privileged` is true performs a transition to `LEADING_GUARD`. After waiting through a delay equal to the guard length it branches to `TRANSMIT` and starts the transfer. When its transmission is complete the sending station changes state to `END_PACKET` and then `HEAR_EOT` as in normal mode. The act of resetting `is_competing` in the first state removes the station as a tournament participant. For controlled mode the latter state enforces an inter-packet gap and then transfers control to `END_SLOT`. The purpose of `END_SLOT` is the same as for an idle slot. The new state `LEADING_GUARD` introduced here has the code:

```
case LEADING_GUARD:
    wait_event (DELAY, guard_length, TRANSMIT);
    return;
```

State `HEAR_EOT` handles both modes as follows:

```
case HEAR_EOT:
    if (the_station->normal_mode)
        wait_event (DELAY, gap_length, IDLE);
    else
        wait_event (DELAY, gap_length, END_SLOT);
    return;
```

Listening stations that hear channel activity in NEW_SLOT again branch to CHANNEL_BUSY. This state prevents a slot timeout in NEW_SLOT. When the end of packet reaches listening stations, they also branch to HEAR_EOT. From there, they have the same route as transmitting stations, to END_SLOT and then IDLE or NEW_SLOT. The last case in NEW_SLOT is when two or more stations use a slot. Stations transfer to HEAR_COLLISION upon hearing the inevitable collision. The privileged stations are in state TRANSMIT when this occurs and unprivileged stations in CHANNEL_BUSY. As with normal mode collisions, this state synchronizes the network and adjusts the reservation schedule via local variables defer_count and delay_count. Functions set_defer_count and set_delay_count consider whether the collision occurred in normal or controlled mode to return appropriate values.

Let us now examine two protocols, DP and TCR, that are special cases of this generic implementation. To complete their implementations, specific versions of functions set_defer_count and set_delay_count as well as supporting information will be provided.

8.3. The DP Protocol Implementation

The DP protocol implementation is completed with its versions of set_defer_count and set_delay_count. For this protocol, stations require one additional local variable priority to keep track of the station with highest priority which is initialized to 0. Collisions occur only in normal mode so the two functions are simplified. Function set_defer_count allocates one slot per station as given below:

```
void set_defer_count ()
{
    the_station->defer_count = n_stations;
}
```

Function set_delay_count indicates the number of slots until the station becomes privileged. It is based on variable priority that indicates the station privileged during the first slot. Assuming a right circular rotation for slot privilege this function is:

```

void set_delay_count () {
    int station_id;
    station_id = station_to_id (the_station);
    the_station->delay_count = (n_stations + station_id - the_station->priority) % n_stations;
}

```

Variable priority is rotated after each collision, this is accomplished by adding the following code to state HEAR_COLLISION after delay_count has been set.

```

the_station->priority = (the_station->priority + 1) % n_stations

```

This completes the DP specification in terms of the generic semi-controlled implementation. The next example considers a more complex protocol in which controlled mode collisions are possible.

8.4. The TCR Protocol Implementation

The protocol specific aspect of TCR is the virtual network tree. The binary representation of station identifiers operates as the basis for this binary tree. A binary digit of 0 (1) places the station in the left (right) subtree. In general a station is a member of a subtree T_{xy} if the last x bits of its identifier are the binary reversal of y . For example in T_{21} , the level x equals 2 and the two binary digits of y are 01. This subtree contains all stations whose binary identifier terminates with 10.

Consider the implementation of function set_defer_count. It sets the number of remaining controlled slots following a collision. With TCR, collisions can occur in both modes so this function considers the current mode. In normal mode, two slots are allocated after a collision. If controlled mode collisions occur, the function should append 2 additional slots. However for controlled mode collisions stations avoid control state END_SLOT which removes the slot in which the collision occurred. Hence the value of defer_count is only incremented by 1 for this mode. The code for set_defer_count is given by:

```

void set_defer_count () {
    if (the_station->normal_mode)
        the_station->defer_count = 2;
    else
        the_station->defer_count++;
}

```

Function `set_delay_count` is more complex. After normal mode collisions, stations in the left (right) subtree of T_{00} have a delay of 0 (1) slot. When a collision occurs in controlled mode with T_{xy} privileged, the two new slots ($T_{x+1,2y}$ and $T_{x+1,2y+1}$) precede existing slots. Hence, stations in the left (right) subtree of T_{xy} have `delay_count` set to 0 (1); this is the first rule for `set_delay_count`. One way to implement this rule is for stations to maintain the current privileged subtree as was performed by observers described previously. However, it contradicts the local view of protocol entities and introduces a needless calculational burden to implementations.

A simpler method notes that privileged stations have `delay_count` equal to 0 when collisions occur. This indicates T^* is one of the station's ancestors as stations not meeting this condition must wait for the termination of the sub-tournament within this subtree. For example, if station 0 participates in a collision, the privileged subtree is T_{x0} for some x . To uniquely determine the privileged subtree a local counter ply is maintained. It is initialized to 0 when a collision occurs in normal mode and incremented each time the privileged subtree is set to a deeper node. Then, if a station participates in a collision and ply equals x , the current privileged subtree is the station ancestor at level x . Once the subtree T_{xy} is uniquely determined, privileged stations must determine if they are members of $T_{x+1,2y}$ or $T_{x+1,2y+1}$. This is a simple test of the x^{th} low order identifier bit. For example, if T_{10} is the privileged subtree then all privileged stations have a 0 low order bit. The stations in the left (right) subtree of T_{10} have a 0 (1) in bit position 1. Note that T_{10} becomes privileged after 1 collision which is the bit position tested in station identifiers.

The second rule for calculating `delay_count` involves stations whose value of this counter is non-zero. These stations did not participate in the collision and are not within the privileged subtree T_{xy} . Recall that the two new slots for $T_{x+1,2y}$ and $T_{x+1,2y+1}$ precede all existing slots. It appears that these values of `delay_count` should be incremented by 2. However, the slot in which the collision occurred has not yet been removed so `delay_count` is only incremented by 1. This produces the following code:

```
int left_subtree (int level) {
    int station_id;
    station_id = station_to_id (the station);
    return ((station_id >> level) % 2 == 0);
}
```

```

void set_delay_count () {
    int station_id;
    station_id = station_to_id (the_station);
    if (delay_count == 0)
        if (left_subtree (station_id, the_station->ply))
            the_station->ply++;
        else
            the_station->delay_count = 1;
    else
        the_station->delay_count++;
}

```

Variable `ply` is incremented for privileged stations following collisions but remains unchanged for non-privileged stations. Consider its effect following an uncontrolled collision. All stations have `delay_count` initialized to 0 prior to calling `set_delay_count`. The stations in T_{10} (T_{11}) pass (fail) the test performed by `left_subtree`. Upon incrementing `ply`, stations in T_{10} view the privileged subtree as T_{10} , whereas those in T_{11} still view it as T_{00} . However, stations in T_{11} are barred channel access until their value of `delay_count` decreases to 0 which occurs after completion of the T_{10} sub-tournament. When this occurs, they increment `ply` which fixes T^* as T_{11} . Sub-tournaments must terminate on a non-collision slot so a line of code is added to state `END_SLOT` for this purpose. This line checks if `delay_count` has decreased to 0 in a participating station and increments `ply` when this occurs. This process insures that all stations have a current view of T^* when it is their turn to transmit.

8.5. Simulation Results with the TCR Implementation

The TCR implementation presented piecewise in the previous sections and summarized in Appendix E was simulated using LANSF. Three networks were considered with 2, 8 and 32 stations. Stations were uniformly distributed along the channel at separations of 10 time units (bits) to produce channel lengths of 10, 70 and 310 bits. Stations had local clock errors with a maximum value ($\epsilon_{\max}$) of 0.02. The synchronization period Δ^* and slot length Δ were set to their minima to stress test the implementation. Recall that Δ^* equals $2^*(L + S)$ and in LANSF the physical signalling delay S is 0. To account for local clock errors and discrete LANSF time, Δ^* was set to $\frac{2L}{1 - \epsilon_{\max}} + 2$. The denominator buffers against fast local clocks and insures stations have a sufficient minimum packet length. The additional 2 time units

eliminate the problem of simultaneous events being delivered in an incorrect order at either end of intervals. For example, if a collision occurs at the last possible instant, the transmitting station receives simultaneous end of packet and collision events. LANSF delivers these events randomly and a protocol error results with either order. If the end of packet is delivered first the transmitting station incorrectly believes its transfer was successful. When the collision event is first delivered, an end of packet event is pending but not accepted by state HEAR_COLLISION which produces an unspecified response.

The guard length was set to $\frac{\epsilon_{\max}(M_s \Delta + G)}{1 - \epsilon_{\max}} + 2$ where M_s is the maximum number of consecutive idle slots and G the inter-packet gap. The denominator $(1 - \epsilon_{\max})$ accounts for second order effects ignored in the linear approximation. The numerical values in simulated networks were as follows:

Network	2L	Δ^*	M_s	Γ	Δ
1	20	22	0	4	30
2	140	144	2	9	162
3	620	634	4	55	744

The network traffic was uniformly distributed between stations with total interstation intercommunication. Traffic density was set to provide a high load without overloading the network which produces a situation where all stations are backlogged.

Two sets of simulations were performed on all 3 networks. The first set implemented observer `check_privilege` which tested the first claimed property (chapter 6). Another set implementing observers `check_participant` and `check_tournament` tested the latter two properties. Both sets of simulations detected no errors with simulation runs of 100,000 packets on all networks.

Based on these results, implementation correctness relative to formal observer service properties has been proven for the explored states. This of course does not imply correctness for all protocol states. However, it demonstrates that the implementation has improved using observer testing and the observer model for protocol re-implementation. This supports the thesis title, "Developing CSMA/CD Protocols with LANSF Observers". As the title suggests, some protocols may well be too complex to completely verify. As in many engineering problems, it may be the case that improving the product is more realistic

than trying to make it perfect. Further, if the view expressed in section 1.3 about the nature of CSMA/CD errors is valid, then correct simulations of this length strongly imply a correct implementation. The issue of how to continue the CSMA/CD development and testing cycle is considered in the next chapter.

Chapter 9

Conclusions

This thesis examined the problem of verifying a CSMA/CD-TCR protocol implementation. The well accepted methods used in OSI-FDT specification and testing were found to be insufficient for this protocol class. This was related to two CSMA/CD features; the need for a channel description technique and an arbitrary number of protocol entities. Formal Description Techniques do not contain a channel model and their proof techniques are oriented towards two party protocols. The verification problem was then shifted to Local Area Network Simulation Facility (LANSF). Given that LANSF has been successfully used to specify and simulate CSMA/CD protocol implementations, could it also be used to test implementation correctness? A recently added feature called observers offered promise in this direction.

With observer tools to monitor LANSF simulations and an existing implementation of CSMA/CD-TCR, the author set out to determine if the initial implementation was correct. This led to the first question, correct with respect to what standard? In a comparatively mature field such as FDT specifications, there is considerable literature regarding standards to which protocol specifications or implementations should conform. Since LANSF has been used far less than FDTs, there is no existing body of conformance standards. Some standards to which implementations should conform are self-evident. These are general or syntactic protocol properties which include the absence of locks, overspecification or underspecification as well as liveness guarantees. The LANSF modelling system is quite well designed to handle some errors in this category. Simulation results such as channel throughput provide additional diagnostic information.

The more difficult problem however is demonstrating that implementations conform to semantic properties, that is protocol specific features. The author chose to formally specify protocol service properties and use these as standards for semantic conformance testing. Since the uncontrolled TCR mode is well known from other CSMA/CD protocols, there were no service properties explicitly tested within this mode. For the retransmission or controlled mode, three general service properties of the semi-

controlled CSMA/CD subclass were first stated informally. Using these statements three observers were programmed, one to test each property dynamically during LANSF simulations. Each observer corresponded to a formal specification of a protocol service property.

The reason for formally specifying a standard to test an implementation is the same with LANSF or FDTs. These standards can be used mechanically to verify the correctness of large state spaces. With LANSF, a random state exploration was performed by the implementation while observers asserted service properties. The size of these explorations was quite beyond those feasible without mechanical assistance. In total, two incidents of implementation non-conformance were revealed by observers. Of equal or greater value was the underlying observer model, this provided a global temporal framework in which to reason about protocol timing properties. This point is quite significant, previous reasoning about timing correctness was generally performed using a local point of view. This leads to difficult circular correctness proofs. An alternate viewpoint is to reason linearly about the correctness of a global model and show that protocol entities conform to this model.

The protocol was re-implemented using the observer model as a reference. Three networks described in the previous chapter were simulated and no discrepancies were discovered. Developing CSMA/CD protocols with observers appears to be a success from the experiments conducted here. As indicated, the second TCR implementation was conformance tested without any detected errors. It was also possible to algebraically prove implementation timing constraints within the observer time frame. Before concluding that this implementation is correct, two additional questions must be answered affirmatively, these are:

- a. Do the observers test all protocol properties and are they correctly specified?
- b. Was the entire protocol state space examined to insure total correctness?

Unfortunately neither question can be answered with an unqualified yes, it is only possible to present arguments that support this response. To start, the properties of uncontrolled mode were not formally tested. These however are not of a global nature and can be demonstrated within the reference frame of a single protocol entity. For controlled mode three service properties were formalized, these appear to cover the essence of semi-controlled CSMA/CD protocols. However, it should be noted that other protocol

development techniques also lack exhaustive descriptions of service properties. Regarding the question of observer correctness, it can be stated that observers were a second independent and simplified protocol specification. The conformance between an independent observer and implementation suggests that both are correct. Non-conformance usually indicates an error in the more complex specification, which is the implementation. As for the state space that was explored, this problem is examined further in the next section. Future research that can follow from this work is considered in the second section.

9.1. Random State Exploration

The entire question of random state exploration for protocol validation or verification is an open question. Some papers have examined the fault coverage of event sequences based on test suites, cf. [DAHB88], [SIBH88]. This however is not directly applicable to states visited on the basis of random events generated in LANSF. In the case of random event generation, general statistical principles can be used as a starting point. Consider a system with X global states, where testing examines x states in a random, memoryless fashion and the test states contain no errors. The probability of a state not being explored or equivalently the fraction of states not explored is $e^{-x/X}$. For subsequent operation, the average length of random operation before entering an unexplored test state roughly equals the test length of x states. Thus the minimum time of correct operation should be on the order of x states. This result is not very comforting as pointed out earlier, a test of 50,000 CSMA/CD packets corresponds to a few seconds of real time.

An issue of more interest is to what extent the correct operation of tested states implies correctness of unexplored states. Stated in another fashion, do the same errors occur repeatedly in different states or are errors unique? If protocols tend to have a few conceptual errors that manifest themselves repeatedly, then their detection and subsequent correction in a tested state should remove this same error from unexplored states. With regards to this question, experimental results give reason for hope. A paper by West [WEST86] and experiments for this thesis both suggest widespread errors from a few common origins to be the case. In his paper West states:

When a complex protocol is validated using state-exploration techniques, the majority of the errors detected are found many times over in different states of the system. In the recent validation of the OSI Session Layer, approximately 3% of the system states explored manifested an error. In a typical run, exploring 25,000 reachable states, individual errors were detected on average about 40 times.

Testing with LANSF observers leads to similar conclusions. The interface and semantic errors reported in Chapter 6 occurred after short simulation runs, typically less than 100 packets. In these experiments, observers terminated the simulation after locating an error so it cannot be stated when errors would re-appear. However, it is reasonable to speculate that if the simulation had continued, the next error occurrence would have been quite soon. After correcting the original implementation for each error, simulation runs of 100,000 packets did not re-detect the same error.

9.2. Future Directions

This thesis demonstrated a technique for verifying CSMA/CD implementation correctness relative to a standard of service specification. These standards were informally stated and their formalization (ie. observer coding) were performed manually. Such a technique can be expected to have similar problems as discussed in the introduction for protocols preceding the FDT era. This includes an ambiguous interpretation of informal specifications and translation errors in observer coding. Future research could produce a formal language in which to specify protocol properties. Afterwards, automatic translation tools could be implemented to generate observer code. However, previous work by Groz [GROZ86] with Véda has suggested that restricted formal languages are not well suited to express service properties. Their observer language, like that of LANSF, was also a general programming language.

A second area of future research is suggested by developments with Formal Description Techniques. For OSI protocols, the implementation is preceded by an abstract specification of protocol entities. This permits a formal logical analysis prior to implementation. A similar mechanism should be constructed for CSMA/CD and other MAC level protocols. This of course requires a Formal Description Technique that

contains concepts relevant to this sublayer. If such a FDT is produced, it would presumably be possible to formally assert service properties from these "FDT" specifications. To round off the list of tools, an automatic generation of LANSF code from such specifications would also be desirable.

The formalization of the MAC sublayer as described above is a long term research goal. Short term research goals should be centered around the conformance testing of real protocols. The case study of this thesis, TCR, is a good example of a hardware implementable protocol. However the simplifications used within this study ignore important details of the protocol environment. One example is physical layer hardware errors, some of which are considered in Suda et al. [SUDA90]. For future conformance testing, the simulation environment should incorporate "hardware" errors. Protocol implementations and the observer model should consider such events. On this front, there are plans to extend LANSF so that it will permit the simulation of random channel errors.

A second environment detail that should be considered is the continuous arrival of new stations. As users log onto networks, it should be assumed that the station does not know the current network status. With Ethernet this is not a problem, stations remain in uncontrolled mode until their packet collides. Hence it is quite legitimate for new stations to transmit whenever a packet is buffered and they have deferred to current channel activity. However with TCR there may be a tournament in progress; unexpected transmissions by new stations would disrupt the tournament and the protocol might not recover from such events. One solution is to force new stations to wait until the network is in uncontrolled mode. For example, a silent period of more than $\log_2 N$ slots guarantees uncontrolled mode as there cannot be that many consecutive idle slots in controlled mode. Forcing new stations to wait for such periods of silence before transmitting has good and bad points. The bad point is that starvation can occur; the new station may never have an opportunity to transmit. However, if the network is so heavily loaded that it does not have such periods of silence, it may be better to ban new stations until the load decreases.

If we consider it unacceptable to ban new stations, then old network stations should accept the responsibility of helping new stations synchronize. For example, when transmitting packets it can be obligatory to include a preamble such as the number of remaining tournament slots. With uncontrolled transmissions, this field would be set to 0.

However, the whole issue of transmissions by new stations may well be moot if the problem of hardware errors is addressed. A new station transmitting randomly may indeed disrupt a tournament but this should not lock protocols robust enough to handle hardware errors. Such transmissions can simply be viewed as the result of a previous hardware error where the new station did not correctly synchronize because of misinformation from its physical layer. If the protocol does demonstrate such robustness, then the issue of whether or not old stations should help synchronize new stations is a question of efficiency. The overhead of "synchronization" information in packets must be weighed against time lost through needless collisions produced by new stations.

In summary, future research should minimize discrepancies between simulated and hardware environments. In this situation, a LANSF observed implementation would be more easily accepted as a reliable protocol. The hardware implementation would then be little more than taking the user specified LANSF processes and placing them in ROM on a card with a previously implemented hardware layer.

Bibliography

- [802.3] IEEE: 802.3: Carrier Sense Multiple Access with Collision Detection. New York: IEEE, 1985a.
- [ABRA85] N. Abramson. Development of the ALOHANET. *IEEE Trans. on Information Theory*, vol. 31, pp. 119-123, 1985.
- [AYAC79] J. M. Ayache, P. Azéma, M. Diaz. Observer: A Concept for On-Line Detection of Control Errors in Concurrent Systems.
- [BELI89] F. Belina, D. Hogrefe. The CCITT Specification and Description Language SDL. *Computer Networks and ISDN Systems*, vol. 16(4), pp. 311-341, 1989.
- [BOCH88] G. v. Bochmann, J. Vaucher. Adding Performance Aspects to Specification Languages. *Protocol specification, testing and verification*, VIII, pp. 19-31, 1988.
- [BOCH90] G. v. Bochmann. Protocol specification for OSI. *Computer Networks and ISDN Systems*, vol. 18(3), pp. 167-184, 1990.
- [BOLO87] T. Bolognesi, E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, vol. 14(1), pp. 25-59, 1987.
- [BUDK87] S. Budkowski, P. Dembinski. An Introduction to Estelle. A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, vol. 14(1), pp. 3-23, 1987.
- [CAPE79] J. I. Capetanakis. Tree Algorithms for Packet Broadcast Channels. *IEEE Trans on Information Theory*, vol. 25(5), pp. 505-515, 1979.
- [CARP89] B. Carpenter. Is OSI Too Late? *Computer Networks and ISDN Systems*, vol. 17(4&5), pp. 284-286, 1989.
- [DAHB88] A. Dahbura, K. K. Sabnani. An Experience in Estimating Fault Coverage of a Protocol Test. 1988 IEEE Infocom, pp. 71-79.
- [DAY83] J. Day and H. Zimmerman. The OSI reference model. *Proceedings of the IEEE*, vol. 71, pp. 1334-1340, Dec 83.
- [DOBO88] W. Dobosiewicz and P. Gburzynski. Ethernet with segmented carrier. *IEEE Proceedings of Computer Networking Symposium*, pp. 72-78, 1988.
- [DOBO89] W. Dobosiewicz and P. Gburzynski. Improving fairness in CSMA/CD networks. *IEEE Proceedings of SICON '89*, 1989.
- [DSSO86] R. Dssouli, G. v. Bochmann. Conformance testing with multiple observers. *Protocol specification, testing and verification*, VI, pp. 217-229, 1986.
- [ESTE89] ISO IS9074, Estelle: A formal description technique based on an extended state transition model, 1989.
- [ETHE80] The Ethernet. A Local Area Network. Data Link Layer and Physical Layer Specifications. Version 1.0. Published by DEC, Intel, Xerox.

- [GBUR89a] P. Gburzynski, P. Rudnicki. On Executable Specifications, Validation, and Testing of MAC-Level Protocols.
Protocol specification, testing and verification, IX, pp. 261-273, 1989.
- [GBUR89b] P. Gburzynski, P. Rudnicki. The LANSF Protocol Modeling Environment, version 2.0. Technical Report TR 89-19, University of Alberta, 1989.
- [GROZ86] R. Groz. Unrestricted Verification of Protocol Properties on a Simulation Using an Observer Approach.
Protocol specification, testing and verification, VI, pp. 255-266, 1986.
- [ISO82] ISO IS 7498, Reference Model for OSI, 1982.
- [JARD88] C. Jard, J-F Monin, R. Groz. Development of Veda, a Prototyping Tool for Distributed Algorithms.
IEEE Trans. Software Engineering, 14(3), pp. 339-352, 1988.
- [LINN90] R. J. Lynn Jr. Conformance Testing for OSI Protocols.
Computer Networks and ISDN Systems, vol. 18(3), pp. 203-219, 1990.
- [LOTO89] ISO IS8807, LOTOS: A Formal Description Technique, 1989.
- [MAXE87] K. Maxemchuk, K. Sabnani. Probabilistic Verification of Communication Protocols.
Protocol specification, testing and verification, VII, pp. 307-320, 1987.
- [METC76] R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks.
Communications of the ACM, vol. 19(7) pp. 395-404, 1976.
- [MOLV85] R. Molva, M. Diaz, J. M. Ayache. Observer: A Run-Time Checking Tool for Local Area Networks.
Protocol specification, testing and verification, V, pp. 495-506, 1985.
- [MURP89] S. Murphy, P. Gunningberg, J. Kelly. Implementing Protocols with Multiple Specifications: Experiences with Estelle, LOTOS and SDL.
Protocol specification, testing and verification, IX, 1989.
- [PARR88] J. Parrow. Verifying a CSMA/CD-protocol with CCS.
Protocol specification, testing and verification, VIII, pp. 373-384, 1988.
- [PEHR90] B. Pehrson. Protocol Verification for OSI.
Computer Networks and ISDN Systems, vol. 18(3), pp. 185-201, 1990.
- [RAYN87] D. Rayner. OSI Conformance Testing.
Computer Networks and ISDN Systems, vol. 14(1), pp. 79-98, 1987.
- [ROBE72] L. Roberts. Extensions of Packet Communication Technology to a Hand Held Personal Terminal.
Proc. SJCC, pp. 295-298, 1972.
- [RUDI88] H. Rudin. Protocol Engineering: A Critical Assessment.
Protocol specification, testing and verification, VIII, pp. 3-16, 1988.
- [SDL88] CCITT Recommendation Z.100, 1988. SDL - System Description and Definition Language.
- [SIDH88] D. Sidhu, T. K. Leung. Fault Coverage of Protocol Test Methods.
1988 IEEE Infocom, pp. 80-85.

- [SIDH90] D. Sidhu, T. Blumer. Semi-automatic Implementation of OSI protocols.
Computer Networks and ISDN Systems, vol. 18(3), pp. 221-238, 1990.
- [SUDA90] T. Suda, J. J. Bae, D. C. Baxter. The robustness and performance of Tree Collision Resolution algorithms in an unshared feedback error environment.
Computer Networks and ISDN Systems, vol. 18(4), pp. 275-292, 1990.
- [SUNS79] C. Sunshine. Formal Techniques for Protocol Specification and Verification.
Computer, pp. 20-26, 1979.
- [TANE88] A. S. Tanenbaum. Computer Networks, second edition, Chapter 3.
Prentice Hall, 1988.
- [VEDA89] Simulator Observation: Principles and Practices Employed in the VEDA Simulator.
Technical Note NT/LAA/SLC/269, National Center of Telecommunications Studies,
France, 1989.
(Title translated from French to English by the thesis author.)
- [WEST86] C. H. West. Protocol Validation by Random State Exploration.
Protocol specification, testing and verification, VI, pp. 233-242, 1986.

Appendix A

Services used by station processes in CSMA/CD Protocols

TIMER A1

Immediate service:

current_time

Return the current simulation time in this variable.

Future service (wake-up call):

wait_event (DELAY, delay_period, new_state)

In this call, delay_period is the delay until the transition should occur to state new_state. Parameter delay_period is type int.

Macros:

#define continue_at (new_state)

wait_event (DELAY, 0, new_state);
return

#define skip_and_continue_at (new_state)

wait_event (DELAY, 1, new_state);
return

CLIENT A1

Immediate service:

get_packet (BUFFER, min, max, frame_info)

BUFFER is the structure in which the packet is placed (if available). Parameters min and max are the limits on the packet length and frame_info specifies the additional information bits on the frame. The return value of this function indicates whether or not a packet was available.

Future service (wake-up call)

wait_event (CLIENT, MESSAGE_ARRIVAL, new_state)

This service request produces a station transition to new_state when the client produces a packet.

LINK A1

Immediate services (no response)

```
start_transfer (port_id, packet);  
stop_transfer (port_id);  
abort_transfer (port_id);  
start_jam (port_id);  
stop_jam (port_id);
```

In these calls, `port_id` is the station port through which the communication occurs. Variable `packet` is a structure of type `PACKET`.

Future service (wake-up call)

```
wait_event (port_id, LINK_STATUS, new_state)
```

This service concerns the link connected to the station port indexed by `port_id`. It produces a transition to `new_state` when the link status matches that indicated by `LINK_STATUS`. Note that the first parameter is not a single constant as for the two other A1s. The other A1s have large constant values so all small values for the first parameter are assumed to identify ports and the request is directed to the LINK A1.

Macros:

```
#define transmit_packet (port_id, packet, new_state)
```

```
start_transfer (port_id, packet);  
wait_event (DELAY, packet.total_length, new_state)
```

```
#define emit_short_jam (port_id, jam_length, new_state)
```

```
start_jam (port_id);  
wait_event (DELAY, jam_length, new_state)
```

Appendix B

LANSF Ethernet transmitter

```

#define BUS 0
#define BUFFER packet_buffer(0)
#define slot_length 512

transmitter() {
    int min_packet, max_packet, info_length;
    int jam_length, last_silence, idle_period, backoff();

    switch (the_action) {
        case INITIALIZE:
        case IDLE:
            if (!get_packet(BUFFER, min_packet, max_packet, info_length)) {
                wait_event(CLIENT, MESSAGE_ARRIVAL, IDLE);
                return;
            }
            the_station->collision_counter = 0;

        case GAP:
            if (undef(last_silence = last_eoa_sensed(BUS))) {
                wait_event(BUS, SILENCE, GAP);
                return;
            }
            if (idle_period = current_time - last_silence < gap_length) {
                wait_event(DELAY, gap_length - idle_period, TRANSMIT);
                return;
            }

        case TRANSMIT:
            transmit_packet(BUS, BUFFER, END_PACKET);
            wait_event(BUS, COLLISION, HEAR_COLLISION);
            return;

        case END_PACKET:
            stop_transfer(BUS);
            release_packet(BUFFER);
            continue_at(IDLE);

        case HEAR_COLLISION:
            abort_transfer(BUS);
            the_station->collision_counter++;
            emit_short_jam(BUS, jam_length, STOP_JAM);
            return;

        case STOP_JAM:
            stop_jam(BUS);
            wait_event(DELAY, backoff(), GAP);
            return;
    }
}

int backoff() {
    int mback, cc;

    if (cc = the_station->collision_counter > 10) cc = 10;
    mback = 1;
    mback = mback << cc;
    return (slot_length * l_uniform(0, mback));
}

```

Appendix C

LANSF CSMA/CD-TCR transmitter

(Initial implementation)

```
#define BUS 0
#define BUFFER packet_buffer(0)
#define transfer_pending flag_set(BUFFER->flags, BUFFER_FULL)
#define slot_length 512
#define loser_delay 512
#define loser_wait(n) n*loser_delay

int min_packet, max_packet, info_length;
int jam_length, last_silence, idle_period, loser_delay;

transmitter() {
    switch (the_action) {
        case INITIALIZE:
        case IDLE:
            the_station->tournament_in_progress = NO;
            if (transfer_pending) get_packet(BUFFER, min_packet, max_packet, info_length);
            if (transfer_pending) {
                wait_event(BUS, ACTIVITY, CHANNEL_BUSY);
                wait_event(CLIENT, MESSAGE_ARRIVAL, IDLE);
                return;
            }
            if (undef(last_silence = last_eoa_sensed(BUS))) continue_at(CHANNEL_BUSY);
            if (idle_period = current_time - last_silence < gap_length) {
                wait_event(DELAY, gap_length - idle_period, TRANSMIT);
                return;
            }
        case TRANSMIT:
            the_station->active = YES;
            transmit_packet(BUS, BUFFER, END_PACKET);
            wait_event(BUS, COLLISION, HEAR_COLLISION);
            return;
        case CHANNEL_BUSY:
            wait_event(BUS, EOT, END_PACKET);
            wait_event(BUS, COLLISION, HEAR_COLLISION);
            return;
        case HEAR_COLLISION:
            if (the_station->active) {
                abort_transfer(BUS);
                the_station->active = NO;
                emit_short_jam(BUS, jam_length, STOP_JAM);
            } else {
                wait_event(DELAY, jam_length + collision_delay, COLLISION_GONE);
            }
            return;
        case STOP_JAM:
            stop_jam(BUS);
            wait_event(DELAY, collision_delay, COLLISION_GONE);
            return;
    }
}
```

```

case COLLISION_GONE:
    if (the_station->tournament_in_progress) {
        the_station->tournament_in_progress = YES;
        the_station->ply = 0;
        the_station->delay_count = 0;
        the_station->defer_count = 1;
    }
    the_station->defer_count++;
    if (transfer_pending) {
        if (the_station->delay_count == 0) {
            the_station->ply++;
            if (left_subtree(the_station->ply) continue_at (TRANSMIT);
        }
        the_station->delay_count++;
        wait_event (DELAY, loser_wait (the_station->delay_count), REJOIN);
        wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
        return;
    }
    wait_event (DELAY, loser_wait (the_station->defer_count, DEFERRED);
    wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
    return;

case END_PACKET:
    if (the_station->tournament_in_progress) {
        the_station->defer_count--;
        if (the_station->active) {
            stop_transfer (BUS);
            release_packet (BUFFER);
            the_station->active = NO;
            wait_event (DELAY, gap_length + loser_wait (the_station->defer_count), DEFERRED);
            wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
            return;
        }
        if (transfer_pending) {
            the_station->delay_count--;
            wait_event (DELAY, gap_length + loser_wait (the_station->delay_count), REJOIN);
            wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
            return;
        }
        wait_event (DELAY, gap_length + loser_wait (the_station->defer_count), DEFERRED);
        wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
        return;
    }
    else {
        if (the_station->active) {
            stop_transfer (BUS);
            release_packet (BUFFER);
            the_station->active = NO;
        }
        continue_at (IDLE);
    }

case REJOIN:
    the_station->delay_count = 0;
    wait_event (DELAY, rejoin_delay, TRANSMIT);
    return;

case DEFERRED:
    wait_event (DELAY, rejoin_delay, IDLE);
    return;

```

Appendix D

LANSF CSMA/CD-TCR observers

```
#define LEFT 0
#define RIGHT 1
#define max_depth 6
#define max_stations 32

int tree_depth, *tree;
int participant [max_depth] [max_stations];
int privileged_station ();
void deeper_subtree (), next_subtree ();

check_privilege () {
    switch (the_observer_action) {
        case INITIALIZE:
            tree_depth = 0;
            tree = (int *) memreq (max_depth * sizeof(int));
            resume_at (SLOT_STARTING);

        case SLOT_STARTING:
            timeout (slot_length, SLOT_FINISHED);
            inspect (ANY, transmitter, ANY, TRANSMIT, STATION_BOT);
            return;

        case SLOT_FINISHED:
            next_subtree ();
            resume_at (SLOT_STARTING);

        case STATION_BOT:
            if (!privileged_station ())
                excptn ("An unprivileged station is transmitting.");
            inspect (ANY, transmitter, ANY, TRANSMIT, STATION_BOT);
            inspect (ANY, transmitter, ANY, END_PACKET, STATION_EOT);
            inspect (ANY, transmitter, ANY, HEAR_COLLISION, START_COLLISION);
            return;

        case STATION_EOT:
            if (uncontrolled ())
                resume_at (SLOT_FINISHED);
            else {
                timeout (gap_length, SLOT_FINISHED);
                return;
            }

        case START_COLLISION:
            collision_cleared = current_time + jam_length + slot_length;
            resume_at (CLEAR_COLLISION);

        case CLEAR_COLLISION:
            timeout (collision_cleared - current_time, DEEPER_SUBTREE);
            inspect (ANY, transmitter, ANY, TRANSMIT, REPORT_BOT);
            return;

        case REPORT_BOT:
            if (!privileged_station ())
                excptn ("An unprivileged station is transmitting.");
            resume_at (CLEAR_COLLISION);

        case DEEPER_SUBTREE:
            deeper_subtree ();
            resume_at (SLOT_STARTING);
    }
}
```

```

int uncontrolled () { return (tree_depth == 0); }

void deeper_subtree () { tree [tree_depth++] = LEFT; }

void next_subtree () {
    if (tree_depth > 0) {
        if (tree [tree_depth - 1] == LEFT) {
            tree [tree_depth - 1] = RIGHT;
        } else {
            tree_depth--;
            next_subtree ();
        }
    }
}

int bit_value (int an_integer, position) {
    an_integer = an_integer >> position;
    return (an_integer % 2);
}

int privileged_station () {
    int index, station_id;

    station_id = station_to_id (the_station);
    for (index = 0; index < tree_depth; index++)
        if (bit_value (station_id, index) != tree [index])
            return (FALSE);
    return (TRUE);
}

void log_participant () {
    int station_id;

    station_id = station_to_id (the_station);
    participant [tree_depth] [station_id] = TRUE;
}

void clear_participant () {
    int depth, station_id;

    station_id = station_to_id (the_station);
    if (participant [tree_depth] [station_id])
        for (depth = 0; depth <= tree_depth; depth++)
            participant [depth] [station_id] = FALSE;
    else
        exceptn ("An unregistered station is finishing a transmission.");
}

int level_empty (int the_level) {
    int station_id;

    for (station_id = 0; station_id < n_stations; station_id++)
        if (participant [the_level] [station_id]) return (FALSE);
    return (TRUE);
}

```

Appendix E LANSF CSMA/CD-TCR transmitter (Final implementation)

```

#define BUS 0
#define BUFFER packet_buffer(0)

int min_packet, max_packet, info_length, jam_length;
int synch_period, guard_length, max_idle_slots, slot_length;
float max_clock_error;

transmitter () {
    switch (the_action) {
        case INITIALIZE:
            the_station->normal_mode = TRUE;
            the_station->is_transmitting = FALSE;
            the_station->is_competing = FALSE;
            wait_event (DELAY, gap_length, IDLE);
            return;

        case IDLE:
            if (get_packet (BUFFER, min_length, max_length, info_length)) {
                continue_at (TRANSMIT);
            } else {
                wait_event (CLIENT, MESSAGE_ARRIVAL, IDLE);
                wait_event (BUS, ACTIVITY, CHANNEL_BUSY);
                return;
            }

        case TRANSMIT:
            the_station->is_transmitting = TRUE;
            the_station->is_competing = TRUE;
            transmit_packet (BUS, BUFFER, END_PACKET);
            wait_event (BUS, COLLISION, HEAR_COLLISION);
            return;

        case CHANNEL_BUSY:
            wait_event (BUS, EOT, HEAR_EOT);
            wait_event (BUS, COLLISION, HEAR_COLLISION);
            return;

        case END_PACKET:
            the_station->is_transmitting = FALSE;
            the_station->is_competing = FALSE;
            stop_transfer (BUS);
            release_packet (BUFFER);
            continue_at (HEAR_EOT);

        case HEAR_EOT:
            if (the_station->normal_mode)
                wait_event (DELAY, gap_length, IDLE);
            else
                wait_event (DELAY, gap_length, END_SLOT);
            return;
    }
}

```

```

case HEAR_COLLISION:
    if (the_station->is_transmitting) {
        abort_transfer(BUS);
        the_station->is_transmitting = FALSE;
    }
    if (the_station->normal_mode) {
        the_station->delay_count = 0;
        the_station->ply = 0;
    }
    set_defer_count();
    if (the_station->is_competing) set_delay_count();
    the_station->normal_mode = FALSE;
    emit_short_jam(BUS, jam_length, STOP_JAM);
    return;

case STOP_JAM:
    stop_jam(BUS);
    wait_event(DELAY, slot_length, NEW_SLOT);
    return;

case NEW_SLOT:
    if (station_privileged()) {
        continue_at(LEADING_GUARD);
    } else {
        wait_event(DELAY, slot_length, END_SLOT);
        wait_event(BUS, ACTIVITY, CHANNEL_BUSY);
        return;
    }

case END_SLOT:
    the_station->defer_count--;
    if (the_station->defer_count == 0) {
        the_station->normal_mode = TRUE;
        wait_event(DELAY, guard_length, IDLE);
        return;
    } else {
        if (the_station->is_competing) {
            the_station->delay_count--;
            if (the_station->delay_count == 0) the_station->ply++;
        }
        continue_at(NEW_SLOT);
    }

case LEADING_GUARD:
    wait_event(DELAY, guard_length, TRANSMIT);
    return;
}

}

int station_privileged()
{ return (the_station->is_competing && the_station->delay_count != 0); }

void set_defer_count() {
    if (the_station->normal_mode)
        the_station->defer_count = 2;
    else
        the_station->defer_count++;
}

```

```

void set_delay_count () {
    int station_id;

    station_id = station_to_id (the_station);
    if (the_station->delay_count == 0)
        if (left_subtree (station_id, the_station->ply))
            the_station->ply++;
        else
            the_station->delay_count = 1;
    else
        the_station->delay_count++;
}

int left_subtree (int level) {
    int station_id;

    station_id = station_to_id (the_station);
    return ((station_id >> level) % 2 == 0);
}

```